2019

Measuring And Securing Cryptographic Deployments

Luke Taylor Valenta
University of Pennsylvania, luke.valenta@gmail.com

Measuring And Securing Cryptographic Deployments

Abstract

This dissertation examines security vulnerabilities that arise due to communication failures and incentive mismatches along the path from cryptographic algorithm design to eventual deployment. I present six case studies demonstrating vulnerabilities in real-world cryptographic deployments. I also provide a framework with which to analyze the root cause of cryptographic vulnerabilities by characterizing them as failures in four key stages of the deployment process: algorithm design and cryptanalysis, standardization, implementation, and endpoint deployment. Each stage of this process is error-prone and influenced by various external factors, the incentives of which are not always aligned with security. I validate the framework by applying it to the six presented case studies, tracing each vulnerability back to communication failures or incentive mismatches in the deployment process.

To curate these case studies, I develop novel techniques to measure both existing and new cryptographic attacks, and demonstrate the widespread impact of these attacks on real-world systems through measurement and cryptanalysis. While I do not claim that all cryptographic vulnerabilities can be described with this framework, I present a non-trivial (in fact substantial) number of case studies demonstrating that this framework characterizes the root cause of failures in a diverse set of cryptographic deployments.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Nadia A. Heninger

Keywords

Cryptography, Diffie-Hellman, Elliptic curves, Internet scanning, TLS

Subject Categories

Computer Sciences

MEASURING AND SECURING CRYPTOGRAPHIC DEPLOYMENTS

Luke Valenta

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2019

Supervisor of Dissertation

Nadia Heninger, Adjunct Associate Professor of Computer and Information Science

Graduate Group Chairperson

Rajeev Alur, Zisman Family Professor and Computer and Information Science Graduate Group Chair

Dissertation Committee

Jonathan M. Smith, Professor of Computer and Information Science

Matthew A. Blaze, Adjunct Professor of Computer and Information Science

Boon Thau Loo, Professor of Computer and Information Science

Matthew D. Green, Associate Professor at the Johns Hopkins Information Security Institute

ACKNOWLEDGEMENT

One page is not enough room to list all of the collaborators, mentors, staff, classmates, and friends to which I am grateful for making the last five years so enjoyable, enlightening, and interesting. I am especially grateful to those below:

Nadia Heninger, not only for her valuable guidance and support, but for the adventures—from acro yoga in the park to parties on pirate ships—that made this journey unforgettable. Prof. Smith, for always being there to provide wisdom and advice, and working tirelessly to help others. Prof. Loo, for his encouragement and enthusiastic support throughout my graduate school career. Prof. Blaze, who is always willing to share his vast knowledge and experience. Prof. Green, for his insight, thoughtfulness, willingness to collaborate, and exceptional ability to spark research ideas. Brandi Adams, for convincing me to give research a try, and David Levin for helping me to discover the joys of research.

Nick Roessler, for all of the thought-provoking conversations, board game nights, and Philly adventures we've shared over these past five years. I couldn't ask for a better friend, roommate, and juggling partner. Marcella Hastings, for being an amazing labmate, travel companion, and all around one of the most fun and personable people I know. Shaanan Cohney, who is a genuine and truly wonderful friend, always willing to loop me in as a co-conspirator in mischief. Rafi Ruben, for patiently teaching me everything I know about sysadmining, and for always being willing to drop everything to help a friend.

My parents, Van and Claudia, for reading my papers despite claiming they "didn't understand a word," and for never failing to provide me with advice and love when I need it. My siblings, Sarah, Stephen, David, Lydia, Peter, and Abigail, who have inspired me with their own accomplishments and continue to be some of my best friends in the world. Finally, I am grateful to Nicole for her continuous love and support, and for giving me so many reasons to be excited about my next chapter in life.

ABSTRACT

MEASURING AND SECURING CRYPTOGRAPHIC DEPLOYMENTS

Luke Valenta

Nadia Heninger

This dissertation examines security vulnerabilities that arise due to communication failures and incentive mismatches along the path from cryptographic algorithm design to eventual deployment. I present six case studies demonstrating vulnerabilities in real-world cryptographic deployments. I also provide a framework with which to analyze the root cause of cryptographic vulnerabilities by characterizing them as failures in four key stages of the deployment process: algorithm design and cryptanalysis, standardization, implementation, and endpoint deployment. Each stage of this process is error-prone and influenced by various external factors, the incentives of which are not always aligned with security. I validate the framework by applying it to the six presented case studies, tracing each vulnerability back to communication failures or incentive mismatches in the deployment process.

To curate these case studies, I develop novel techniques to measure both existing and new cryptographic attacks, and demonstrate the widespread impact of these attacks on real-world systems through measurement and cryptanalysis. While I do not claim that *all* cryptographic vulnerabilities can be described with this framework, I present a non-trivial (in fact substantial) number of case studies demonstrating that this framework characterizes the root cause of failures in a diverse set of cryptographic deployments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1 : Introduction

Billions of users every day rely on cryptographic protocols like Transport Layer Security (TLS), Secure Shell (SSH), and Internet Protocol Security (IPsec) for communication security. The security of these protocols relies both on their theoretical cryptographic underpinnings and their concrete instantiations. Over the years, cryptographic primitives have evolved to adhere to stronger notions of security as the field advances. However, despite decades of innovation and constant improvement to the state of the art, the cryptographic security of real-world systems is not a solved problem. Cryptographic primitives, standards, and implementations have failed repeatedly over the years, leaving a trail of vulnerable systems. New attacks are constantly being developed, and old attacks resurface in unexpected ways, driving the need for continual reevaluation of the security of deployed systems.

In this dissertation, I present six research studies that uncover both new and existing attacks in deployed cryptographic systems. To understand the provenance of these vulnerabilities, I introduce a novel framework for characterizing cryptographic vulnerabilities as communication failures and misaligned incentives in the multi-stage cryptographic deployment process, and use this framework to trace each of the presented case studies back to their root causes.

1.1. A framework for analyzing failures in cryptographic deployments

In this section, I break down the cryptographic deployment process into four primary stages, and describe the factors that influence each stage. Understanding each stage is a crucial for determining the origination of cryptographic failures in real-world deployments.

1.1.1. Algorithm design and cryptanalysis

The first step in the cryptographic deployment process is algorithm design. Cryptographic algorithm designers are typically cryptography experts that have deep understandings of the mathematical underpinnings of their constructions. In this stage, the algorithm designer constructs a cryptographic primitive and (often) proves its security based on mathematical assumptions in some adversarial model. Low-level details such as encoding formats, random number generation, and exact parameter instantiations are abstracted in order to focus on

the mathematical details. However, algorithm designers may provide guidelines for choosing concrete parameters based on the asymptotic complexity of the best known attacks against the primitive.

1.1.2. Standardization

The next stage in the cryptographic deployment framework is standardization. Standards development is driven by trusted standards bodies such as National Institute of Standards and Technology (NIST), the IEEE Standards Association (IEEE-SA), and the Internet Engineering Task Force (IETF). These organizations develop standards for government, industry, and private use. Standards organization are concerned with improving existing standards as well as developing new standards to adapt to the evolving needs of technology users. This involves identifying cryptographic algorithms for standardization and providing detailed guidelines for their use in applications to ensure interoperability. The standardization process also allows for scrutiny of cryptographic primitives by the security community and makes standardized options available for common cryptographic operations to help prevent non-experts from "rolling their own crypto."

While algorithm designers and cryptanalysts may have some say in the cryptographic standardization process, there are many other players involved, each with their own incentives. These parties include governments, vendors implementing the standards, security professionals, and the end users of the technology. Thus, the standardization process must cater to many external requests and demands, and usually results in a final specification for which security is only one of many priorities such as interoperability, regulatory compliance, and implementation flexibility.

1.1.3. Implementation

Following standardization, implementation is the next stage of cryptographic deployment. The outputs of this stage are software or hardware implementations of cryptographic protocols and primitives ready to be deployed in applications. Implementers of cryptographic libraries are often individuals or teams of programmers with some specialized knowledge of the protocols and algorithms to be implemented. Standards often leave many choices to

implementations regarding programming language, performance optimizations, algorithm support, and parameter selection. For example, IETF standards designate where implementation choices are required with key words like "MUST", "SHOULD", and "MAY" in protocol specifications [76]. Implementations may seek accreditation of compliance with standards requirements such as NIST's FIPS 140-2 standard [233], or may skip standards-compliance altogether.

In general, implementations are only as secure as the protocols they implement, and are susceptible to additional human errors and programming mistakes. Further, cryptographic implementations must be concerned with attack vectors such as side-channel attacks that may be out of scope for standards specifications, while providing the performance demanded by users of the implementation.

Cryptographic libraries expose APIs that are then used by application developers to build products used in real-world cryptographic deployments. Library developers should not assume that application developers have the same in-depth understanding of the cryptographic primitives, and should thus expose simple and robust APIs to prevent misuse.

1.1.4. Endpoint deployment

Endpoint deployment is the final step of the deployment process. Application developers and system administrators are the main parties involved in this stage.

Application developers choose cryptographic implementations to meet some desired functionality within larger applications. For example, an application that requires a secure communication channel may use a TLS implementation such as OpenSSL to provide this functionality. Application developers are responsible for correctly using library APIs to carry out some task, and providing the configurations necessary for a secure deployment.

System administrators determine the exact hardware, software, and configuration of endpoint cryptographic deployments. They are responsible for updating applications to incorporate security updates, bug fixes, and new features, and for keeping systems running

efficiently and effectively. While security is often a requirement, it may not be prioritized as highly as availability, functionality, compliance, and performance. System administrators may also have time constraints and organizational pressures that prevent them from properly configuring, updating, and maintaining their systems.

While some system administrators and application developers do have in-depth knowledge of cryptographic algorithm design, this is often not the case and should not be assumed. Thus, cryptographic implementations should clearly describe the risk accompanying the available performance, functionality, and security trade-offs that they provide to their users, and provide secure-by-default configurations.

1.1.5. Background

I now provide background on recurring themes that appear in the presented case studies.

Diffie-Hellman key exchange. Diffie-Hellman key exchange is a public-key algorithm used as a fundamental building block for many cryptographic protocols. It allows two parties to derive a shared secret after seeing each other's public keys, with no prior communication necessary. In the basic protocol, Alice and Bob each generate private keys a and b and transmit their public keys g^a and g^b, respectively. Upon seeing the other party's public key, Alice and Bob can each compute the value g^{ab} as a shared secret. If the Diffie-Hellman parameters and public keys are chosen carefully, the basic protocol is secure against a passive eavesdropper who can observe the public values.

Small subgroup attacks and public key validation. When this protocol is deployed in real-world systems, additional care must be taken to ensure security. For instance, the basic protocol trivially breaks in the presence of an active Monster-in-the-Middle (MitM) attacker if messages between Alice and Bob are not properly authenticated. Further, if Bob uses the same secret key b for multiple connections, Mallory may be able to carry out an attack against connection with other parties.

Several attacks take advantage of the fact that Bob must perform operations that involve

his secret key using an untrusted input from his peer. A class of attacks known as *small subgroup attacks* are possible if Bob uses a maliciously chosen public key from Mallory to compute the shared secret. The attacks can either force the derived shared secret to be a predictable value, or leak information about Bob's secret key through a side channel or from the key exchange output. Such attacks can be prevented if Bob performs *key exchange validation* to detect and reject invalid public keys. The specific checks required for key exchange validation vary depending on the domain parameters, and may require some level of computation.

Although the attacks and defenses have been known for decades, there are many reasons why implementations might omit validation, including for performance, a reduction in code complexity, or an assumption that the attacks will be prevented by some other protocol mechanism. However, a break in these assumptions can lead to devastating attacks. The work presented in this dissertation shows that lack of validation continues to be a source of attacks against Diffie-Hellman in real-world systems.

Cryptographic agility. Modern protocols are often designed to have *cryptographic agility* [170], meaning that it should be possible for the protocol to easily migrate from older, less secure algorithms to newer, more secure algorithms. Since implementations are not updated all at once, but gradually over time, up-to-date implementations that support the latest cryptographic parameters must still support the outdated algorithms if they wish to communicate with older clients and servers. When standards do not provide sufficient agility, implementations tend to resort to *ad hoc* negotiation mechanisms, such as repeatedly falling back to a previous protocol version until both parties indicate support [225].

Obsolete and intentionally weakened cryptography. Perhaps the most well-known examples of obsolete cryptography lurking in modern systems are the export-grade ciphers in the SSL/TLS protocols. Throughout the 1990s, U.S. law severely restricted the export of cryptographic devices and algorithms from the United States. When the SSL protocol was initially developed in 1995, International Traffic in Arms Regulations (ITAR) [242] pro-

hibited the export of "Information Security Systems and equipment, cryptographic devices, software, and components", with an eventual exception made for weak, breakable encryption. To comply with the regulations, early versions of SSL included cipher suites that limited the security to 512-bit public keys for RSA and Diffie-Hellman key exchange, and 40-bit keys for symmetric ciphers. The export restrictions were gradually lifted through the year 2000, and starting with TLSv1.1 export cipher suites were no longer included in TLS standards. However, implementations often continue to support export-grade cryptography for backwards compatibility with existing systems.

Downgrade attacks. Maintaining support for obsolete and weakened cryptography in modern systems can come with adverse side effects. First, backwards compatibility removes the pressure to update legacy systems that are old and broken, leaving them exposed to attacks. Second, a modern system maintaining support for weakened and vulnerable protocols or parameters can increase its attack surface, in some cases exposing itself to unexpected attack vectors. In particular, it invites downgrade attacks, where a MitM attacker can force two communicating parties to use security parameters weaker than they would have otherwise negotiated.

1.2. Case studies

This section provides overviews of each of the six cryptographic vulnerability case studies contributed by this dissertation, and demonstrates how each can be characterized using the provided cryptographic deployment framework.

Table 1, which is referenced throughout the case study summaries in this section, shows how each vulnerability can be characterized. The table indicates which stages of the cryptographic deployment process are primarily responsible, and provides explanations for the origins of each weakness.

The table columns providing explanations for each weakness are as follows: *Mistake* means that the weakness is the result of a human error such a programming mistake or misconfiguration. *Misunderstanding* represents a conscious decision (as opposed to a mistake) to

Ch.	No.	Weakness	Design/Cryptanalysis	Standardization	Implementation	Deployment	Mistake	Misunderstanding	Interoperability	Performance	Compliance	Simplicity
2	1	Reuse FFDH private key		X	X	X				X		
	2	Fail to validate FFDH subgroup order		X	X	X	X	X		X		X
	3	Use short subgroup for FFDH (non-safe prime)		X	X	X		X				
	4	Fail to specify FFDH subgroup order		X								X
	5	Use short exponents for FFDH			X	X				X		
	6	Expose security-critical choices to developers			X				X	X		
3	7	Fail to sign ECDH group (allows downgrade)		X			X					
	8	Support weak ECDH groups		X	X	X		X	X			
	9	Fail to validate ECDH subgroup order		X	X	X	X	X		X		X
	10	Reuse ECDH private key		X	X	X				X		
4	11	Non-constant time X25519 implementation			X		X					
	12	Fail to validate X25519 subgroup order	X	X	X		X	X				X
5	13	Standardize export (512-bit) RSA		X							X	
	14	Continue to support export (512-bit) RSA			X	X		X	X			
6	15	Advances in discrete log cryptanalysis	X	X	X	X		X				
	16	Use only a few, widely shared DH primes	X	X	X	X		X	X			X
	17	Fail to sign ciphersuite (allows downgrade)		X			X					
	18	Standardize export DHE ciphersuites		X							X	
	19	Continue to support export DHE ciphersuites			X	X		X	X			
7	20	Use RSA with PKCS#1 v1.5 padding		X					X			
	21	Support SSLv2			X	X	X	X	X			
	22	Share RSA keys across deployments				X						X
	23	Extra clear bug			X	X	X					
	24	Fail to sign ciphersuite (allows downgrade)		X			X					

Table 1: **Characterizing cryptographic attacks**—Attacks against cryptographic deployments can be traced back to a mistake, misunderstanding, or incentive mismatch in the deployment process.

introduce the weakness without a clear understanding of the security risks. This could be the result of a failure to communicate security requirements between stages of deployment, or the result of new advances in cryptanalysis that fundamentally change our understanding of the security of cryptographic primitives. *Interoperability* means that the decision to introduce the weakness was made to allow for backwards compatibility with existing deployments, or to allow deployments flexibility to adapt in the future. *Performance* means that a weakness was introduced to improve some aspect of performance. *Compliance* decisions are made in order to comply with regulatory requirements; the introduction of export-grade cryptography into the SSL/TLS standard is a prime example. *Simplicity* represents a weakness that is introduced because it allows for less complex (and hopefully less error-prone) deployments, or because it makes deployment easier.

Not all of the weaknesses described in Table 1 constitute exploitable vulnerabilities in isolation, but they do when combined with other weaknesses. The fact that many of these attacks presented in the case studies require multiple weaknesses to be exploitable in practice demonstrates a lack of defense in depth in deployed cryptographic systems.

1.2.1. Finite field Diffie-Hellman attacks and measurements

Chapter 2 presents a measurement survey of small subgroup attacks in finite field Diffie-Hellman (FFDH) deployments. The case study first explores the impact of these attacks against the TLS, SSH, and IPsec protocols. We examined over 20 cryptographic libraries to understand their implementation choices, and found that until January 2016, *none* performed the requisite validation to prevent small subgroup attacks by default. We found full key recovery attacks against OpenSSL, the Exim mail server, and the Unbound DNS resolver. We then used ZMap [114] to perform scans of the IPv4 Internet to measure server behavior regarding Diffie-Hellman parameter choice, key reuse, and key exchange validation. We found that many server performed insufficient validation for the Diffie-Hellman parameters that they supported, potentially leaving them open to attack. Finally, we characterized the complexity of small subgroup attacks against Diffie-Hellman groups found in the wild to understand the security provided by these groups when used by implementations

that fail to perform validation.

Vulnerability characterization. I now discuss how small subgroup attacks, which have been known about for decades, make their way into modern FFDH cryptographic deployments. Small subgroup key recovery attacks in the FFDH setting require a deployment to reuse the same private key for multiple connections, use a non-safe prime, and fail to perform key exchange validation.

Security proofs for protocols using Diffie-Hellman often require that private keys are ephemeral to ensure forward secrecy and to prevent key leakage through repeated small subgroup attacks. However, generating fresh ephemeral Diffie-Hellman keys for each new connection can be computationally expensive, especially for servers that handle high workloads. Thus, system administrators prefer to cache and reuse ephemeral values to improve performance (**No. 1**). This study found that this behavior was enabled by default for some library implementations (e.g., OpenSSL), and was common for servers on the Internet. Some standards explicitly forbid this practice, while others place no restrictions on ephemeral key reuse [218].

Diffie-Hellman requires that public keys are non-identity elements of a sufficiently large subgroup. To ensure this, standards specify one set of validation checks for safe prime groups, and another set of validation checks for non-safe primes in "DSA-style" groups that includes an additional exponentiation to check the subgroup order. The validation checks required for safe prime groups are simpler and less computationally expensive, so several standards including IPsec and SSH required implementations to use safe prime groups only, and specified the simpler validation checks (**No. 2**). When non-safe primes were later standardized for IETF protocols including TLS, SSH, and IPsec [206] (**No. 3**), this required implementations using these parameters to perform additional validation checks to avoid small subgroup attacks. However, protocols like SSH and TLS that allow servers to unilaterally specify the group parameters during the key exchange protocol do not provide a way for the server to specify the group order to allow the client to perform the additional

checks (**No. 4**). Thus, deployments of these protocols that were configured to use these non-safe primes were unable to perform the necessary validation (**No. 2**) and became vulnerable to small subgroup attacks if configured to reuse private keys. Full key recovery attacks are made possible when implementations additionally use short exponents (e.g., 128 bits for a 1024-bit prime group) to match the size of the group's expected security (**No. 5**).

I hypothesize that many practical attacks are made possible because tradeoffs between security and performance are left to the deployment stage, potentially forcing non-experts to make security-critical decisions that require domain-specific knowledge (**No. 6**). Without a proper understanding of the known risks associated with those tradeoffs at endpoint deployments, it is no surprise that security vulnerabilities arise.

This chapter is based on work [297] published at NDSS 2017 in collaboration with David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger.

1.2.2. Elliptic curve Diffie-Hellman attacks and measurements

Chapter 3 presents a survey of the protocol-level and implementation-level attack surface of elliptic curve Diffie-Hellman (ECDH) in TLS, SSH, IPsec, and JSON Web Encryption (JWE). The study starts by analyzing the feasibility of a theoretical downgrade attack called CurveSwap that allows a MitM adversary to downgrade the security of a TLS connection to the weakest Diffie-Hellman group supported by both client and server, as long as the MitM can forge the TLS Finished MAC before the connection times out. The study also gauges the feasibility of similar curve downgrade attacks against SSH and IPsec. Contributions of this work include extensive active and passive measurements of implementation choices regarding algorithm support, key reuse, and validation to determine if any client and server deployments were vulnerable to the downgrade attack. Through our scans, we found a significant population of servers failing to properly validate key exchange parameters, as well as a large number of servers repeating key exchange values across connections. However, we did not find any servers exhibiting both behaviors, which would have led to invalid curve

key recovery attacks. We also examined source code and found several implementations of the JWE standard vulnerable to classic invalid curve attacks, stemming from the fact that the standard neglects to mention that curve validity checks are necessary.

Vulnerability characterization. The CurveSwap attack itself is possible due to a flaw in the TLS standard (**No. 7**). However, we found widespread support for weak (but not quite practically exploitable) elliptic curve groups in many client and server deployments that increase the seriousness of the downgrade attack. This can be explained by the pressure on standards and implementations to provide cryptographic agility to allow deployments to adapt to their speed and security needs. However, supporting weak parameters can be a liability, as the CurveSwap attack demonstrates (**No. 8**).

Similar to the study in Chapter 2 of FFDH deployments, this study finds that a significant number of ECDH deployments fail to perform proper key exchange validation for the negotiated group. The TLS, SSH, and IPsec standards all *do* properly specify the validation checks required of implementations. However, we found that a small percentage of deployments failed to implement the checks (**No. 9**). While the study did find that a ECDH ephemeral key reuse is very prevalent in TLS, SSH, and IPsec (**No. 10**), it did not find any deployments that both reused ephemeral keys and failed to perform proper key exchange validation. However, in the case of JWE, where Diffie-Hellman public keys are reused, the standards fail to mention validation entirely, resulting in full invalid curve key recovery attacks against several implementations.

This chapter is based on work [298] published at EuroS&P 2018 in collaboration with Nick Sullivan, Antonio Sanso, and Nadia Heninger.

1.2.3. Side-channel attack on Curve25519 using low-order elements

Curve25519 is an elliptic curve designed to be resistant to many known attacks, while also allowing for simple, side-channel resistant implementations. Users of Curve25519 are advised that input validation is unnecessary, since the function is designed to process all inputs without error and without leaking information about the private key from the output of

the function. Chapter 4 presents a cache side-channel attack against Libgcrypt's implementation of Curve25519 that is made possible though a combination of non-constant time arithmetic and lack of input validation. The attack uses an "order-4" element on Curve25519 to trigger side-channel leakage when passed in to Libgcrypt's ECDH decryption operation. Had the implementation performed input validation, these malicious inputs would have been rejected. We demonstrate full key recovery attacks against three applications that use the vulnerable library: encrypted git, email, and messaging.

Vulnerability characterization. Despite using the side-channel resistant Montgomery ladder algorithm with the Curve25519 function that is designed to resist key recovery attacks, this study demonstrates full key recovery attacks against Libgcrypt's Curve25519 implementation.

The attack works as follows: when a low-order element is passed into the Montgomery ladder scalar-by-point multiplication routine, it introduces a key-dependent mathematical structure in the computation; with the ability to measure this structure through a side-channel attack, one can extract the bits of the private key. Thus, the root cause of the vulnerability in Libgcrypt's implementation is the non-constant time arithmetic that exposes a mathematical structure through side-channel leakage (**No. 11**). The leakage can then be measured using a cache-based side-channel to complete the attack. However, practical exploitation of the vulnerability was possible because the implementation failed to perform input validation to reject low-order elements, following the recommendation of the Curve25519 designer [56] (**No. 12**).

This chapter is based on joint work with Daniel Genkin and Yuval Yarom, and was published at CCS 2017 [145].

1.2.4. 512-bit RSA in the wild

Chapter 5 examines the security impact and prevalence of weak RSA public keys in the wild. We focus on 512-bit RSA keys, which were exploited in the FREAK attack to MitM TLS connections [61]. To demonstrate the ease with which 512-bit RSA keys can be factored

with modern computing capabilities, we optimized and parallelized the Cado-NFS implementation of the number field sieve (NFS) algorithm and performed fine-grained benchmark tests using rented cloud computing resources from Amazon EC2. We were able to reliably factor 512-bit RSA keys in under 4 hours for a cost of $75. We then performed a survey of RSA key sizes across several protocols, and found that 512-bit RSA keys are surprisingly persistent in deployed systems, with hundreds of keys found in deployments of DNSSEC, HTTPS, IMAP, POP3, SMTP, DKIM, SSH, and PGP.

Vulnerability characterization. This study finds that 512-bit RSA keys are surprisingly persistent in cryptographic deployments. RSA keys of this size were known to be insecure even before their standardization in SSL for compliance with export regulations (**No. 13**), but decades later, a long tail of deployments still have not removed these weak keys from their configurations (**No. 14**). I hypothesize that the root cause is that the knowledge of decades-old advances in cryptanalysis have not been properly propagated all the way to the system administrators responsible for these deployments.

This chapter is based on work [296] published at Financial Cryptography 2016 in collaboration with Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger.

1.2.5. Logjam attack and measurements

Chapter 6 investigates Diffie-Hellman key exchange and finds it to be less secure in practice than was previously believed. We present the Logjam attack, which allows a MitM attacker to downgrade a connection to use a weak, export-grade 512-bit Diffie-Hellman group even when both peers prefer secure parameters. The attack is a protocol flaw in TLS that affects any server that accepts obsolete export-grade DHE cipher suites and any client willing to negotiate weak FFDH groups. The paper demonstrates that computing 512-bit discrete logarithms online for the MitM attack is much easier than was previously thought due to the fact that a large portion of the computation can be precomputed. We then measure the impact of the Logjam attack by performing extensive Internet scans. In particular, we find that 8.4% of the Alexa top 1M websites were vulnerable to the attack. Further, 82%

of the vulnerable servers used the *same* three 512-bit export DH groups, so performing the precomputation for only these three groups would allow us to compromise 7% of Alexa top 1M sites.

Vulnerability characterization. Arguably the most interesting discovery of this work is that Diffie-Hellman is much less secure in practice than previously thought for *all* FFDH deployments. Advances in discrete log cryptanalysis using the NFS algorithm showed that a large precomputation given only the prime is possible, which subsequently allows for very fast individual discrete logarithm computations for groups using that prime (**No. 15**). While the overall asymptotic running time of the algorithm remains the same, the cost of a single precomputation in the group can be amortized over many individual discrete log computations. Despite the possibility for this precomputation, a select few Diffie-Hellman groups were standardized and shared by the majority of hosts on the Internet, suggesting that standards developers were not aware of these cryptanalytic advances (**No. 16**).

The Logjam downgrade attack is possible due to a flaw in the TLS protocol (**No. 17**), along with support in cryptographic deployments for legacy export-grade DHE ciphersuites. Similar to the downgrade attack of Chapter 3, Logjam is a parameter downgrade attack that takes advantage of the fact that TLS parameter negotiation messages can be tampered with by a MitM attacker as long as they are able to forge the TLS Finished MAC. Implementation and deployment support for export-grade DHE ciphersuites makes the attack much more devastating with the majority of hosts using the *same* three Diffie-Hellman groups (**No. 16**). Thus, a single 512-bit discrete log computation can be performed once for a given prime, and then used to break TLS connections online via the Logjam downgrade attack. Government export regulations on cryptography from the 1990s are the root cause of known-weak cryptography being forced into use in old versions of SSL/TLS (**No. 18**), and backwards compatibility is to blame for the prevalence of these known-weak parameters in modern systems (**No. 19**).

This chapter is based on joint work [29] with David Adrian, Karthikeyan Bhargavan, Za-

kir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. The paper won the Best Paper Award at ACM CCS 2015, as well as the Pwnie Award for Most Innovative Research at Black Hat 2015.

1.2.6. DROWN attack and measurements

Chapter 7 discusses the DROWN attack and provides measurements that show its widespread impact due to cryptographic parameter reuse. The DROWN attack is a complex cross-protocol attack that is made possible through a combination of weaknesses, and allows an attacker to break modern TLSv1.2 connections provided that the server shares a public key or certificate with an SSLv2 deployment. There are two main versions of the attack.

The *general* DROWN attack targets previously unnoticed protocol flaws in SSLv2 to develop a Bleichenbacher padding oracle attack. The SSLv2 oracle is then used to decrypt modern TLSv1.2 RSA ciphertexts when the SSLv2 and TLSv1.2 servers share a certificate or public key. The attacker must passively capture 1,000 TLS sessions that use RSA key exchange, make about 40,000 connections to the SSLv2 server, and perform 2^{50} symmetric encryption operations offline. We were able to perform the computation in under 8 hours at a cost of $440 using specialized GPU servers on Amazon EC2 [43]. The general DROWN attack does not rely on any implementation flaws, and takes advantage of commonly supported export-grade symmetric ciphers in SSLv2 implementations. Measurements from 2016 show that 33% of all HTTPS servers were vulnerable to the attack.

The *special* DROWN attack relies on a bug in older versions of OpenSSL that brings the attack complexity down sufficiently far that it can be completed in under a minute on a single CPU—fast enough to allow a successful MitM attack against modern browsers. 26% of HTTPS servers were vulnerable to the special DROWN attack at the time of the attack discovery. The DROWN attack demonstrates the risk of supporting old broken protocols for backwards compatibility. It shows that SSLv2 is not only weak, but actively harmful to the security ecosystem.

Vulnerability characterization. RSA with the PKCS#1 v1.5 padding scheme has been known to be weak for decades since Bleichenbacher's original padding oracle attack in 1998 [73], and suitable replacements with provable security such as RSA-OAEP were proposed. However, instead of updating TLS standards to replace PKCS#1 v1.5 padding with more secure variants, the standards developers instead required library developers to add implementation-level countermeasures (**No. 20**). Possible explanations are that updating the standard would break compatibility with existing implementations, and since Bleichenbacher's original attack was considered impractical (it required one million chosen ciphertexts to carry out), the added security did not justify the expense. Regardless of the reasoning, this demonstrates incentives not aligned with security.

The SSLv2 protocol was shown to be irredeemably flawed soon after its standardization, and quickly replaced by SSLv3. However, implementations maintained support for this broken scheme for backwards compatibility with older clients, favoring interoperability over security (**No. 21**).

The DROWN attack is able to decrypt modern TLS RSA ciphertexts due to the sharing of cryptographic parameters with insecure deployments (**No. 22**). There is no security benefit for such parameter sharing, but it can be explained by convenience and cost reduction for system administrators, as acquiring separate certificates for each offered service could be expensive and inconvenient. The risk of this cryptographic parameter sharing in deployments was not fully understood until demonstrated by this attack.

The special DROWN variant of the attack uses the "extra clear" bug in OpenSSL's SSLv2 implementation to drastically reduce the complexity of the attack (**No. 23**) so that it can be used in an online attack. Using the same ciphersuite downgrade weakness exploited in Chapter 6, a MitM attacker can force a connection to use an RSA key exchange when that is not the preferred key exchange mechanism (**No. 24**), and even substitute the server's certificate with another certificate sharing a common name whose key is exposed via a special DROWN oracle.

This chapter is based on joint work [43] with Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. The paper was a finalist for the Facebook Internet Defense Prize at USENIX Security 2016, and won the Pwnie Award for Best Cryptographic Attack at Black Hat 2016.

CHAPTER 2 : Measuring finite field Diffie-Hellman

2.1. Introduction

Diffie-Hellman key exchange is one of the most common public-key cryptographic methods in use in the Internet. It is a fundamental building block for IPsec, SSH, and TLS. In the textbook presentation of finite field Diffie-Hellman, Alice and Bob agree on a large prime p and an integer g modulo p. Alice chooses a secret integer x_a and transmits a public value $g^{x_a} \bmod p$; Bob chooses a secret integer x_b and transmits his public value $g^{x_b} \bmod p$. Both Alice and Bob can reconstruct a shared secret $g^{x_a x_b} \bmod p$, but the best known way for a passive eavesdropper to reconstruct this secret is to compute the discrete log of either Alice or Bob's public value. Specifically, given g, p, and $g^x \bmod p$, an attacker must calculate x.

In order for the discrete log problem to be hard, Diffie-Hellman parameters must be chosen carefully. A typical recommendation is that p should be a "safe" prime, that is, that $p = 2q + 1$ for some prime q, and that g should generate the group of order q modulo p. For p that are not safe, the group order q can be much smaller than p. For security, q must still be large enough to thwart known attacks, which for prime q run in time $O(\sqrt{q})$. A common parameter choice is to use a 160-bit q with a 1024-bit p or a 224-bit q with a 2048-bit p, to match the security level under different cryptanalytic attacks. Diffie-Hellman parameters with p and q of these sizes were suggested for use and standardized in DSA signatures [239]. For brevity, we will refer to these non-safe primes as DSA primes, and to groups using DSA primes with smaller values of q as DSA groups.

A downside of using DSA primes instead of safe primes for Diffie-Hellman is that implementations must perform additional validation checks to ensure the key exchange values they receive from the other party are contained in the correct subgroup modulo p. The validation consists of performing an extra exponentiation step. If implementations fail to validate, a 1997 attack of Lim and Lee [207] can allow an attacker to recover a static exponent by repeatedly sending key exchange values that are in very small subgroups. We

describe several variants of small subgroup confinement attacks that allow an attacker with access to authentication secrets to mount a much more efficient man-in-the-middle attack against clients and servers that do not validate group orders. Despite the risks posed by these well-known attacks on DSA groups, NIST SP 800-56A, "Recommendations for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography" [49] specifically recommends DSA group parameters for Diffie-Hellman, rather than recommending using safe primes. RFC 5114 [206] includes several DSA groups for use in IETF standards.

We observe that few Diffie-Hellman implementations actually validate subgroup orders, in spite of the fact that small subgroup attacks and countermeasures are well-known and specified in every standard suggesting the use of DSA groups for Diffie-Hellman, and DSA groups are commonly implemented and supported in popular protocols. For some protocols, including TLS and SSH, that enable the server to unilaterally specify the group used for key exchange, this validation step is not possible for clients to perform with DSA primes—there is no way for the server to communicate to the client the intended order of the group. Many standards involving DSA groups further suggest that the order of the subgroup should be matched to the length of the private exponent. Using shorter private exponents yields faster exponentiation times, and is a commonly implemented optimization. However, these standards provide no security justification for decreasing the size of the subgroup to match the size of the exponents, rather than using as large a subgroup as possible. We discuss possible motivations for these recommendations later in the paper.

We conclude that adopting the Diffie-Hellman group recommendations from RFC 5114 and NIST SP 800-56A may create vulnerabilities for organizations using existing cryptographic implementations, as many libraries allow user-configurable groups but have unsafe default behaviors. This highlights the need to consider developer usability and implementation fragility when designing or updating cryptographic standards.

Our Contributions. We study the implementation landscape of Diffie-Hellman from several perspectives and measure the security impact of the widespread failure of imple-

mentations to follow best security practices:

- We summarize the concrete impact of small-subgroup confinement attacks and small subgroup key recovery attacks on TLS, IKE, and SSH handshakes.

- We examined the code of a wide variety of cryptographic libraries to understand their implementation choices. We find feasible full private exponent recovery vulnerabilities in OpenSSL and the Unbound DNS resolver, and a partial private exponent recovery vulnerability for the parameters used by the Amazon Elastic Load Balancer. We observe that *no* implementation that we examined validated group order for subgroups of order larger than two by default prior to January 2016, leaving users potentially vulnerable to small subgroup confinement attacks.

- We performed Internet-wide scans of HTTPS, POP3S, SMTP with STARTTLS, SSH, IKEv1, and IKEv2, to provide a snapshot of the deployment of DSA groups and other non-"safe" primes for Diffie-Hellman, quantify the incidence of repeated public exponents in the wild, and quantify the lack of validation checks even for safe primes.

- We performed a best-effort attempt to factor $p-1$ for all non-safe primes that we found in the wild, using ˜100,000 core-hours of computation. Group 23 from RFC 5114, a 2048-bit prime, is particularly vulnerable to small subgroup key recovery attacks; for TLS a full key recovery requires 2^{33} online work and 2^{47} offline work to recover a 224-bit exponent.

Disclosure and Mitigations. We reported the small subgroup key recovery vulnerability to OpenSSL in January 2016 [273]. OpenSSL issued a patch to add additional validation checks and generate single-use private exponents by default [27]. We reported the Amazon load balancer vulnerability in November 2015. Amazon responded to our report informing us that they have removed Diffie-Hellman from their recommmended ELB security policy, and have reached out to their customers to recommend that they use these latest policies. Based on scans performed in February and May 2016, 88% of the affected hosts appear

to have corrected their exponent generation behavior. We found several libraries that had vulnerable combinations of behaviours, including Unbound DNS, GnuTLS, LibTomCrypt, and Exim. We disclosed to the developers of these libraries. Unbound issued a patch, GnuTLS acknowledged the report but did not patch, and LibTomCrypt did not respond. Exim responded to our bug report stating that they would use their own generated Diffie-Hellman groups by default, without specifying subgroup order for validation [253, 295]. We found products from Cisco, Microsoft, and VMWare lacking validation that key exchange values were in the range $(1, p-1)$. We informed these companies, and discuss their responses in Section 2.3.4.

2.2. Background

2.2.1. Groups, orders, and generators

The two types of groups used for Diffie-Hellman key exchange in practice are multiplicative groups over finite fields ("mod p") and elliptic curve groups. We focus on the "mod p" case, so a group is typically specified by a prime p and a generator g, which generates a multiplicative subgroup modulo p. Optionally, the group order q can be specified; this is the smallest positive integer q satisfying $g^q \equiv 1 \bmod p$. Equivalently, it is the number of distinct elements of the subgroup $\{g, g^2, g^3, \ldots \bmod p\}$.

By Lagrange's theorem, the order q of the subgroup generated by g modulo p must be a divisor of $p-1$. Since p is prime, $p-1$ will be even, and there will always be a subgroup of order 2 generated by the element -1. For the other factors q_i of $p-1$, there are subgroups of order $q_i \bmod p$. One can find a generator g_i of a subgroup of order q_i using a randomized algorithm: try random integers h until $h^{(p-1)/q_i} \neq 1 \bmod p$; $g_i = h^{(p-1)/q_i} \bmod p$ is a generator of the subgroup. A random h will satisfy this property with probability $1 - 1/q_i$.

In theory, neither p nor q is required to be prime. Diffie-Hellman key exchange is possible with a composite modulus and with a composite group order. In such cases, the order of the full multiplicative group modulo p is $\phi(p)$ where ϕ is Euler's totient function, and the order of the subgroup generated by g must divide $\phi(p)$. Outside of implementation mistakes,

Diffie-Hellman in practice is done modulo prime p.

2.2.2. Diffie-Hellman Key Exchange

Diffie-Hellman key exchange allows two parties to agree on a shared secret in the presence of an eavesdropper [108]. Alice and Bob begin by agreeing on shared parameters (prime p, generator g, and optionally group order q) for an algebraic group. Depending on the protocol, the group may be requested by the initiator (as in IKE), unilaterally chosen by the responder (as in TLS), or fixed by the protocol itself (SSH originally built in support for a single group).

Having agreed on a group, Alice chooses a secret $x_a < q$ and sends Bob $y_a = g^{x_a} \bmod p$. Likewise, Bob chooses a secret $x_b < q$ and sends Alice $y_b = g^{x_b} \bmod p$. Each participant then computes the shared secret key $g^{x_a x_b} \bmod p$.

Depending on the implementation, the public values y_a and y_b might be *ephemeral*—freshly generated for each connection—or *static* and reused for many connections.

2.2.3. Discrete log algorithms

The best known attack against Diffie-Hellman is for the eavesdropper to compute the the private exponent x by calculating the discrete log of one of Alice or Bob's public value y. With knowledge of the exponent, the attacker can trivially compute the shared secret. It is not known in general whether the hardness of computing the shared secret from the public values is equivalent to the hardness of discrete log.

The *computational Diffie-Hellman assumption* states that computing the shared secret $g^{x_a x_b}$ from g^{x_a} and g^{x_b} is hard for some choice of groups. A stronger assumption, the *decisional Diffie-Hellman problem*, states that given g^{x_a} and g^{x_b}, the shared secret $g^{x_a x_b}$ is computationally indistinguishable from random for some groups. This assumption is often not true for groups used in practice; even with safe primes as defined below, many implementations use a generator that generates the full group of order $p - 1$, rather than the subgroup of order $(p-1)/2$. This means that a passive attacker can always learn the value of the secret exponent modulo 2. To avoid leaking this bit of information about the exponent, both sides

could agree to compute the shared secret as $y^{2x} \bmod p$. We have not seen implementations with this behavior.

There are several families of discrete log algorithms, each of which apply to special types of groups and parameter choices. Implementations must take care to avoid choices vulnerable to any particular algorithm. These include:

Small-order groups. The Pollard rho [261] and Shanks' baby step-giant step algorithms [279] each can be used to compute discrete logs in groups of order q in time $O(\sqrt{q})$. To avoid being vulnerable, implementations must choose a group order with bit length at least twice the desired bit security of the key exchange. In practice, this means that group orders q should be at least 160 bits for an 80-bit security level.

Composite-order groups. If the group order q is a composite with prime factorization $q = \prod_i q_i^{e_i}$, then the attacker can use the Pohlig-Hellman algorithm [259] to compute a discrete log in time $O(\sum_i e_i \sqrt{q_i})$. The Pohlig-Hellman algorithm computes the discrete log in each subgroup of order $q_i^{e_i}$ and then uses the Chinese remainder theorem to reconstruct the log modulo q. Adrian et al. [29] found several thousand TLS hosts using primes with composite-order groups, and were able to compute discrete logs for several hundred Diffie-Hellman key exchanges using this algorithm. To avoid being vulnerable, implementations should choose g so that it generates a subgroup of large prime order modulo p.

Short exponents. If the secret exponent x_a is relatively small or lies within a known range of values of a relatively small size, m, then the Pollard lambda "kangaroo" algorithm [262] can be used to find x_a in time $O(\sqrt{m})$. To avoid this attack, implementations should choose secret exponents to have bit length at least twice the desired security level. For example, using a 256-bit exponent for for a 128-bit security level.

Small prime moduli. When the subgroup order is not small or composite, and the prime modulus p is relatively large, the fastest known algorithm is the number field sieve [154], which runs in subexponential time in the bit length of p, $\exp\left((1.923 + o(1))(\log p)^{1/3}(\log\log p)^{2/3}\right)$.

Adrian et al. recently applied the number field sieve to attack 512-bit primes in about 90,000 core-hours [29], and they argue that attacking 1024-bit primes—which are widely used in practice—is within the resources of large governments. To avoid this attack, current recommendations call for p to be at least 2048 bits [50]. When selecting parameters, implementers should ensure all attacks take at least as long as the number field sieve for their parameter set.

2.2.4. Diffie-Hellman group characteristics

"Safe" primes. In order to maximize the size of the subgroup used for Diffie-Hellman, one can choose a p such that $p = 2q + 1$ for some prime q. Such a p is called a "safe" prime, and q is a Sophie Germain prime. For sufficiently large safe primes, the best attack will be solving the discrete log using the number field sieve. Many standards explicitly specify the use of safe primes for Diffie-Hellman in practice. The Oakley protocol [249] specified five "well-known" groups for Diffie-Hellman in 1998. These included three safe primes of size 768, 1024, and 1536 bits, and was later expanded to include six more groups in 2003 [193]. The Oakley groups have been built into numerous other standards, including IKE [164] and SSH [311].

DSA groups. The DSA signature algorithm [239] is also based on the hardness of discrete log. DSA parameters have a subgroup order q of much smaller size than p. In this case $p - 1 = qr$ where q is prime and r is a large composite, and g generates a group of order q. FIPS 186-4 [239] specifies 160-bit q for 1024-bit p and 224- or 256-bit q for 2048-bit p. The small size of the subgroup allows the signature to be much shorter than the size of p.

2.2.5. DSA Group Standardization

DSA-style parameters have also been recommended for use for Diffie-Hellman key exchange. NIST Special Publication 800-56A, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography" [49], first published in 2007, specifies that finite field Diffie-Hellman should be done over a prime-order subgroup q of size 160 bits for a 1024-bit prime p, and a 224- or 256-bit subgroup for a 2048-bit prime. While the order of the multiplicative subgroups is in line with the hardness of computing discrete logs in

these subgroups, no explanation is given for recommending a subgroup of precisely this size rather than setting a minimum subgroup size or using a safe prime. Using a shorter exponent will make modular exponentiation more efficient, but the order of the subgroup q does not increase efficiency—on the contrary, the additional modular exponentiation required to validate that a received key exchange message is contained in the correct subgroup will render key exchange with DSA primes less efficient than using a "safe" prime for the same exponent length. Choosing a small subgroup order is not known to have much impact on other cryptanalytic attacks, although the number field sieve is somewhat (not asymptotically) easier as the linear algebra step is performed modulo the subgroup order q. [29]

RFC 5114, "Additional Diffie-Hellman Groups for Use with IETF Standards" [206], specifies three DSA groups with the above orders "for use in IKE, TLS, SSH, etc." These groups were taken from test data published by NIST [238]. They have been widely implemented in IPsec and TLS, as we will show below. We refer to these groups as Group 22 (1024-bit group with 160-bit subgroup), Group 23 (2048-bit group with 224-bit subgroup), and Group 24 (2048-bit group with 256-bit subgroup) throughout the remainder of the paper to be consistent with the group numbers assigned for IKE.

RFC 6989, "Additional Diffie-Hellman Tests for the Internet Key Exchange Protocol Version 2 (IKEv2)" [280], notes that "mod p" groups with small subgroups can be vulnerable to small subgroup attacks, and mandates that IKE implementations should validate that the received value is in the correct subgroup or never repeat exponents.

2.2.6. Small subgroup attacks

Since the security of Diffie-Hellman relies crucially on the group parameters, implementations can be vulnerable to an attacker who provides maliciously generated parameters that change the properties of the group. With the right parameters and implementation decisions, an attaker may be able to efficiently determine the Diffie-Hellman shared secret. In some cases, a passive attacker may be able to break a transcript offline.

Small subgroup confinement attacks. In a small subgroup confinement attack, an attacker (either a man-in-the-middle or a malicious client or server) provides a key-exchange value y that lies in a subgroup of small order. This forces the other party's view of the shared secret, y^x, to lie in the subgroup generated by the attacker. This type of attack was described by van Oorschot and Wiener [300] and ascribed to Vanstone and Anderson and Vaudenay [38]. Small subgroup confinement attacks are possible even when the server does not repeat exponents—the only requirement is that an implementation does not validate that received Diffie-Hellman key exchange values are in the correct subgroup.

When working $\bmod p$, there is always a subgroup of order 2, since $p-1$ is even. A malicious client Mallory could initiate a Diffie-Hellman key exchange value with Alice and send her the value $y_M = p - 1 \equiv -1 \bmod p$, which is is a generator of the group of order 2 $\bmod p$. When Alice attempts to compute her view of the shared secret as $k_a = y_M^a \bmod p$, there are only two possible values, 1 and $-1 \bmod p$.

The same type of attack works if $p - 1$ has other small factors q_i. Mallory can send a generator g_i of a group of order q_i as her Diffie-Hellman key exchange value. Alice's view of the shared secret will be an element of the subgroup of order q_i. Mallory then has a $1/q_i$ chance of blindly guessing Alice's shared secret in this invalid group. Given a message from Alice encrypted using Alice's view of the shared secret, Mallory can brute force Alice's shared secret in q_i guesses.

More recently, Bhargavan and Delignat-Lavaud [67] describe "key synchronization" attacks against IKEv2 where a man-in-the-middle connects to both the initiator and responder in different connections, uses a small subgroup confinement attack against both, and observes that there is a $1/q_i$ probability of the shared secrets being the same in both connections. Bhargavan and Leurent [63] describe several attacks that use subgroup confinement attacks to obtain a transcript collision and break protocol authentication.

To protect against subgroup confinement attacks, implementations should use prime-order

subgroups with known subgroup order. Both parties must validate that the key exchange values they receive are in the proper subgroup. That is, for a known subgroup order q, a received Diffie-Hellman key exchange value y should satisfy $y^q \equiv 1 \bmod p$. For a safe prime, it suffices to check that y is strictly between 1 and $p-1$.

Small subgroup key recovery attacks. Lim and Lee [207] discovered a further attack that arises when an implementation fails to validate subgroup order and resues a static secret exponent for multiple key exchanges. A malicious party may be able to perform multiple subgroup confinement attacks for different prime factors q_i of $p-1$ and then use the Chinese remainder theorem to reconstruct the static secret exponent.

The attack works as follows. Let $p-1$ have many small factors $p-1 = q_1 q_2 \dots q_n$. Mallory, a malicious client, uses the procedure described in Section 2.2.1 to find a generator of the subgroup g_i of order $q_i \bmod p$. Then Mallory transmits g_i as her Diffie-Hellman key exchange value, and receives a message encrypted with Alice's view of the shared secret $g_i^{x_a}$, which Mallory can brute force to learn the value of $x_a \bmod q_i$. Once Mallory has repeated this process several times, she can use the Chinese remainder theorem to reconstruct $x_a \bmod \prod_i q_i$. The running time of this attack is $\sum_i q_i$, assuming that Mallory performs an offline brute-force search for each subgroup.

A randomly chosen prime p is likely to have subgroups of large enough order that this attack is infeasible to carry out for all subgroups. However, if in addition Alice's secret exponent x_a is small, then Mallory only needs to carry out this attack for a subset of subgroups of orders $q_1, \dots, q_k$ satisfying $\prod_{i=0}^{k} q_i > x_a$, since the Chinese remainder theorem ensures that x_a will be uniquely defined. Mallory can also improve on the running time of the attack by taking advantage of the Pollard lambda algorithm. That is, she could use a small subgroup attack to learn the value of $x_a \bmod \prod_{i=1}^{k} q_i$ for a subset of subgroups $\prod_{i=1}^{k} q_i < x_a$, and then use the Pollard lambda algorithm to reconstruct the full value of a, as it has now been confined to a smaller interval.

Application	Crypto Library	Short Exponent	Exponent Reuse
OpenSSH	OpenSSL	No	No
Cerberus	OpenSSL	No	Yes
GNU lsh	GnuTLS	No	No
Dropbear	LibTomCrypt	No	No
Lighttpd	OpenSSL	Yes	No
Unbound	OpenSSL	Yes	Yes
Exim	OpenSSL	Library dependent	Yes
Postfix	OpenSSL	No	No

Table 2: **Common application behavior**—Applications make a diverse set of decisions on how to handle Diffie-Hellman exponents, likely due to the plethora of conflicting, confusing, and incorrect recommendations available.

In summary, an implementation is vulnerable to small subgroup key recovery attacks if it does not verify that received Diffie-Hellman key exchange values are in the correct subgroup; uses a prime p such that $p-1$ has small factors; and reuses Diffie-Hellman secret exponent values. The attack is made even more practical if the implementation uses small exponents.

A related attack exists for elliptic curve groups: an invalid curve attack. Similarly to the case we describe above, the attacker generates a series of elliptic curve points of small order and sends these points as key exchange messages to the victim. If the victim does not validate that the received point is on the intended curve, they return a response that reveals information about the secret key modulo different group orders. After enough queries, the attacker can learn the victim's entire secret. Jager, Schwenk, and Somorovsky [179] examined eight elliptic curve implementations and discovered two that failed to validate the received curve point. For elliptic curve groups, this attack can be much more devastating because the attacker has much more freedom in generating different curves, and can thus find many different small prime order subgroups. For the finite field Diffie-Hellman attack, the attacker is limited only to those subgroups whose orders are factors of $p-1$.

2.3. TLS

TLS (Transport Layer Security) is a transport layer protocol designed to provide confidentiality, integrity and (most commonly) one-side authentication for application sessions. It

is widely used to protect HTTP and mail protocols.

A TLS client initiates a TLS handshake with the `ClientHello` message. This message includes a list of supported cipher suites, and a client random nonce r_c. The server responds with a `ServerHello` message containing the chosen cipher suite and server random nonce r_s, and a `Certificate` message that includes the server's X.509 certificate. If the server selects a cipher suite using ephemeral Diffie-Hellman key exchange, the server additionally sends a `ServerKeyExchange` message containing the server's choice of Diffie-Hellman parameters p and g, the server's Diffie-Hellman public value $y_s = g^{x_s} \bmod p$, a signature by the server's private key over both the client and server nonces (r_c and r_s), and the server's Diffie-Hellman parameters (p, g, and y_s). The client then verifies the signature using the public key from the server's certificate, and responds with a `ClientKeyExchange` message containing the client's Diffie-Hellman public value $y_c = g^{x_c} \bmod p$. The Diffie-Hellman shared secret $Y = g^{x_s x_c} \bmod p$ is used to derive encryption and MAC keys. The client then sends `ChangeCipherSpec` and `Finished` messages. The `Finished` message contains a hash of the handshake transcript, and is encrypted and authenticated using the derived encryption and MAC keys. Upon decrypting and authenticating this message, the server verifies that the hash of the transcript matches the expected hash. Provided the hash matches, the server then sends its own `ChangeCipherSpec` and `Finished` messages, which the client then verifies. If either side fails to decrypt or authenticate the `Finished` messages, or if the transcript hashes do not match, the connection fails immediately [106].

TLS also specifies a mode of using Diffie-Hellman with fixed parameters from the server's certificate [260]. This mode is not forward secret, was never widely adopted, and has been removed from all modern browsers due to dangerous protocol flaws [168]. The only widely used form of Diffie-Hellman in TLS today is ephemeral Diffie-Hellman, described above.

2.3.1. Small Subgroup Attacks in TLS

Small subgroup confinement attacks. A malicious TLS server can perform a variant of the small subgroup attack against a client by selecting group parameters g and p such

that g generates an insecure group order. TLS versions prior to 1.3 give the server complete liberty to choose the group, and they do not include any method for the server to specify the desired group order q to the client. This means a client has no feasible way to validate that the group sent by the server has the desired level of security or that a server's key exchange value is in the correct group for a non-safe prime.

Similarly, a man in the middle with knowledge of the server's long-term private signing key can use a small subgroup confinement attack to more easily compromise perfect forward secrecy, without having to rewrite an entire connection. The attack is similar to the those described by Bhargavan and Delignat-Lavaud [67]. The attacker modifies the server key exchange message, leaving the prime unchanged, but substituting a generator g_i of a subgroup of small order q_i for the group generator and g_i for the server's key exchange value y_s. The attacker then forges a correct signature for the modified server key exchange message and passes it to the client. The client then responds with a client key exchange message $y_c = g_i^{x_c} \bmod p$, which the man-in-the-middle leaves unchanged. The server's view of the shared secret is then $g_i^{x_c x_s} \bmod p$, and the client's view of the shared secret is $g_i^{x_c} \bmod p$. These views are identical when $x_s \equiv 1 \bmod q_i$, so this connection will succeed with probability $1/q_i$. For small enough q_i, this enables a man in the middle to use a compromised server signing key to decrypt traffic from forward-secret ciphersuites with a reasonable probability of success, while only requiring tampering with a single handshake message, rather than having to actively rewrite the entire connection for the duration of the session.

Furthermore, if the server uses a static Diffie-Hellman key exchange value, then the attacker can perform a small subgroup key-recovery attack as the client in order to learn the server's static exponent $x_s \bmod q_i$ for the small subgroup. This enables the attacker to calculate a custom generator such that the client and server views of the shared secret are always identical, raising the above attack to a 100% probability of success.

Small subgroup key recovery attacks. In TLS, the client must authenticate the handshake before the server, by providing a valid `Finished` message. This forces a small sub-

Implementation	RFC 5114 Support	Allows Short Exponents	Reuses Exponents	Validates Subgroup
Mozilla NSS	No	Yes, hardcoded	No	$g \leq 2$
OpenJDK	No	Yes, uses max of p_size / 2 and 384	No	$g \leq 2$
OpenSSL 1.0.2	Yes	Yes, if q set or if user sets a shorter length	Default until Jan '16	Yes, as of Jan '16
BouncyCastle	Yes	No	Application dependent	$g \leq 2$
Cryptlib	No	Yes, uses quadratic curve calculation	Application dependent	$g \leq 2$
libTomCrypt	No	Yes, hardcoded	Application dependent	No
CryptoPP	No	Yes, uses work factor calculation	Application dependent	No
Botan	Yes	Yes, uses work factor calculation	No	No
GnuTLS	Application dependent	Yes, restricts to q_size (max 256)	Application dependent	$g \leq 2$

Table 3: **TLS library behavior**—We examined popular TLS libraries to determine which weaknesses from Section 2.2.6 were present. Reuse of exponents often depends on the use of the library; the burden is on the application developer to appropriately regenerate exponents. Botan and libTomCrypt both hardcode their own custom groups, while GnuTLS allows users to specify their own parameters.

group key recovery attack against TLS to be primarily online. To perform a Lim-Lee small subgroup key recovery attack against a server static exponent, a malicious client initiates a TLS handshake and sends a generator g_i of a small subgroup of order q_i as its client key exchange message y_c. The server will calculate $Y_s = g_i^{x_s} \bmod p$ as the shared secret. The server's view of the shared secret is confined to the subgroup of order q_i. However, since g_i and g generate separate subgroups, the server's public value $y_s = g_s^x$ gives the attacker no information about the value of the shared secret Y_s. Instead, the attacker must guess a value for $x_s \bmod q_i$, and send the corresponding client `Finished` message. If the server continues the handshake, the attacker learns that the guess is correct. Therefore, assuming the server is reusing a static value for x_s, the attacker needs to perform at most q_i queries to learn the server's secret $x_s \bmod q_i$ [207]. This attack is feasible if q_i is small enough and the server reuses Diffie-Hellman exponents for sufficiently many requests.

The attacker repeats this process for many different primes q_i, and uses the Chinese remainder theorem to combine them modulo the product of the primes q_i. The attacker can also use the Pollard lambda algorithm to reconstruct any remaining bits of the exponent [207].

We note that the TLS False Start extension allows the server to send application data before receiving the client's authentication [201]. The specification only allows this behavior for abbreviated handshakes, which do not include a full key exchange. If a full key exchange

were allowed, the fact that the server authenticates first would allow a malicious client to mount a mostly offline key recovery attack.

2.3.2. OpenSSL

Prior to early 2015, OpenSSL defaulted to using static-ephemeral Diffie-Hellman values. Server applications generate a fresh Diffie-Hellman secret exponent on startup, and reuse this exponent until they are restarted. A server would be vulnerable to small subgroup attacks if it chose a DSA prime, explicitly configured the `dh->length` parameter to generate a short exponent, and failed to set `SSL_OP_SINGLE_DH_USE` to prevent repeated exponents. OpenSSL provides some test code for key generation which configures DSA group parameters, sets an exponent length to the group order, and correctly sets the `SSL_OP_SINGLE_DH_USE` to generate new exponents on every connection. We found this test code widely used across many applications. We discovered that Unbound, a DNS resolver, used the same parameters as the tests, but without setting `SSL_OP_SINGLE_DH_USE`, rendering them vulnerable to a key recovery attack. A number of other applications including Lighttpd used the same or similar code with non-safe primes, but correctly set `SSL_OP_SINGLE_DH_USE`.

In spring 2015, OpenSSL added explicit support for RFC 5114 groups [246], including the ability for servers to specify a subgroup order in a set of Diffie-Hellman group parameters. When the subgroup order is specified, the exponent length is automatically adjusted to match the subgroup size. However, the update did not contain code to validate subgroup order for key exchange values, leaving OpenSSL users vulnerable to precisely the key recovery attack outlined in Section 2.3.1.

We disclosed this vulnerability to OpenSSL in January 2016. The vulnerability was patched by including code to validate subgroup order when a subgroup was specified in a set of Diffie-Hellman parameters and setting `SSL_OP_SINGLE_DH_USE` by default [247]. Prior to this patch, any code using OpenSSL for DSA-style Diffie-Hellman parameters was vulnerable to small subgroup attacks by default.

Protocol	Scan Date	Total Hosts	*Number of hosts that use...*			
			Diffie-Hellman	Non-Safe Primes	Static Exponents	Static Exponents and Non-Safe Primes
HTTPS	2/2016	40,578,754	10,827,565	1,661,856	964,356	309,891
POP3S	10/2015	4,368,656	3,371,616	26,285	32,215	25
STARTTLS	10/2015	3,426,360	3,036,408	1,186,322	30,017	932
SSH	10/2015	15,226,362	10,730,527	281	1,147	0
IKEv1	2/2016	2,571,900	2,571,900	340,300	109	0
IKEv2	2/2016	1,265,800	1,265,800	177,000	52	0

Table 4: **IPv4 non-safe prime and static exponent usage**—Although non-safe primes see widespread use across most protocols, only a small number of hosts reuse exponents and use non-safe primes; these hosts are prime candidates for a small subgroup key recovery attack.

Exim [122], a popular mail server that uses OpenSSL, provides a clear example of the fragile situation created by this update. By default, Exim uses the RFC 5114 Group 23 parameters with OpenSSL, does not set an exponent length, and does not set `SSL_OP_SINGLE_DH_USE`. In a blog post, an Exim developer explains that because of "numerous issues with automatic generation of DH parameters", they added support for fixed groups specified in RFCs and picked Group 23 as the default [253]. Exim narrowly avoided being fully vulnerable to a key recovery attack by not including the size of the subgroup generated by q in the Diffie-Hellman parameters that it passes to OpenSSL. Had this been included, OpenSSL would have automatically shortened the exponent length, leaving the server fully vulnerable to a key recovery attack. For this group, an attacker can recover 130 bits of information about the secret exponent using 2^{33} online queries, but this does not allow the attacker to recover the server's 2048-bit exponent modulo the correct 224-bit group order q as the small subgroup orders q_i are all relatively prime to q.

We looked at several other applications as well, but did not find them to be vulnerable to key recovery attacks (Table 2).

2.3.3. Other Implementations

We examined the source code of multiple TLS implementations (Table 3). Prior to January 2016, no TLS implementations that we examined validated group order, even for the well-known DSA primes from RFC 5114, leaving them vulnerable to small subgroup confinement attacks.

Most of the implementations we examined attempt to match exponent length to the perceived strength of the prime. For example, Mozilla Network Security Services (NSS), the TLS library used in the Firefox browser and some versions of Chrome [135, 231], uses NIST's "comparable key strength" recommendations on key management [50] to determine secret exponent lengths from the length of the prime. [229] Thus NSS uses 160-bit exponents with a 1024-bit prime, and 224-bit exponents with a 2048-bit prime. In fall 2015, NSS added an additional check to ensure that the shared secret $g^{x_a x_b} \not\equiv 1 \bmod p$ [230].

Several implementations go to elaborate lengths to match exponent length to perceived prime strength. The Cryptlib library fits a quadratic curve to the small exponent attack cost table in the original van Oorschot paper [300] and uses the fitted curve to determine safe key lengths [159]. The Crypto++ library uses an explicit "work factor" calculation, evaluating the function $2.4n^{1/3}(\log n)^{2/3}$ [181]. Subgroup order and exponent lengths are set to twice the calculated work factor. The work factor calculation is taken from a 1995 paper by Odlyzko on integer factorization [241]. Botan, a C++ cryptography and TLS library, uses a similar work factor calculation, derived from RFC 3766 [160], which describes best practices as of 2004 for selecting public key strengths when exchanging symmetric keys. RFC 3766 uses a similar work factor algorithm to Odlyzko, intended to model the running time of the number-field sieve. Botan then doubles the length of the work factor to obtain subgroup and exponent lengths [210].

2.3.4. Measurements

We used ZMap [114] to probe the public IPv4 address space for hosts serving three TLS-based protocols: HTTPS, SMTP+STARTTLS, and POP3S. To determine which primes servers were using, we sent a `ClientHello` message containing only ephemeral Diffie-Hellman cipher suites. We combined this data with scans from Censys [2] to determine the overall population. The results are summarized in Table 4.

In August 2016, we conducted additional scans of a random 1% sample of HTTPS hosts on the Internet. First, we checked for nontrivial small subgroup attack vulnerability. For

servers that sent us a prime p such that $p-1$ was divisible by 7, we attempted a handshake using a client key exchange value of $g_7 \bmod p$, where g_7 is a generator of a subgroup of order 7. (7 is the smallest prime factor of $p-1$ for Group 22.) When we send g_7, we expect to correctly guess the `PreMasterSecret` and complete the handshake with one seventh of hosts that do not validate subgroup order. In our scan, we were able to successfully complete a handshake with 1477 of 10714 hosts that offered a prime such that $p-1$ was divisible by 7, implying that approximately 96% of these hosts fail to validate subgroup order six months after OpenSSL pushed a patch adding group order validation for correctly configured groups.

Second, we measured how many hosts performed even the most basic validation of key exchange values. We attempted to connect to HTTPS hosts with the client key exchange values of $y_c = 0 \bmod p, 1 \bmod p, -1 \bmod p$. As Table 5 shows, we found that over 5% of hosts that accepted DHE ciphersuites accepted the key exchange value of $-1 \bmod p$ and derived the `PreMasterSecret` from it. These implementations are vulnerable to a trivial version of the small subgroup confinement attacks described in Section 2.3.1, for *any* prime modulus p. By examining the default web pages of many of these hosts, we identified products from several notable companies including Microsoft, Cisco, and VMWare. When we disclosed these findings, VMWare notified us that they had already applied the fix in the latest version of their products; Microsoft acknowledged the missing checks but chose not to

Key Exchange Value	**Support DHE**	**Accepted**
$0 \bmod p$	143.5 K	87
$1 \bmod p$	142.2 K	4.9 K
$-1 \bmod p$	143.5 K	7.6 K
$g_7 \bmod p$	10.7 K	1.5 K

Table 5: **TLS key exchange validation**—We performed a 1% HTTPS scan in August 2016 to check if servers validated received client key exchange values, offering generators of subgroups of order 1, 2 and 7. Our baseline DHE support number counts hosts willing to negotiate a DHE key exchange, and in the case of g_7, if $p-1$ is divisible by 7. We count hosts as "Accepted" if they reply to the `ClientKeyExchange` message with a `Finished` message. For g_7, we expect this to happen with probability 1/7, suggesting that nearly all of the hosts in our scan did not validate subgroup order.

Group			*Host Counts*			
Source	**Prime Size**	**Subgroup Size**	**HTTPS**	**SMTP**	**POP3S**	**SSH**
RFC 5114 Group 22	1024	160	1,173,147	145	86	0
Amazon Load Balancer	1024	160	277,858	0	1	0
JDK	768	160	146,491	671	16,515	0
JDK	1024	160	52,726	2,445	9,510	0
RFC 5114 Group 24	2048	256	3,543	5	0	6
JDK	2048	224	982	12	20	0
Epson Device	1024	< 948	372	0	0	0
RFC 5114 Group 23	2048	224	371	1,140,363	2	0
Mistyped OpenSSL 512	512	497	0	717	0	0
Other Non-Safe Primes	—	—	6,366	41,964	151	275
Safe Primes	—	—	9,165,709	1,850,086	3,345,331	10,730,246
Total			10,827,565	3,036,408	3,371,616	10,730,527

Table 6: **IPv4 top non-safe primes**—Nine non-safe primes account for the majority of hosts using non-safe primes.

include them since they only use safe primes, and adding the checks may break functionality for some clients that were sending unusual key exchange values; and Cisco informed us that they would investigate the issue.

Of 40.6 M total HTTPS hosts found in our scans, 10.8 M (27%) supported ephemeral Diffie-Hellman, of which 1.6 M (4%) used a non-safe prime, and 309 K (0.8%) used a non-safe prime and reused exponents across multiple connections, making them likely candidates for a small subgroup key recovery attack. We note that the numbers for hosts reusing exponents are an underestimate, since we only mark hosts as such if we found them using the same public Diffie-Hellman value across multiple connections, and some load balancers that cycle among multiple values might have evaded detection.

While 77% of POP3S hosts and 39% of SMTP servers used a non-safe prime, a much smaller number used a non-safe prime and reused exponents (¡0.01% in both protocols), suggesting that the popular implementations (Postfix and Dovecot [119]) that use these primes follow recommendations to use ephemeral Diffie-Hellman values with DSA primes.

Table 6 shows nine groups that accounted for the majority of non-safe primes used by hosts in the wild. Over 1.17 M hosts across all of our HTTPS scans negotiated Group 22 in a key exchange. To get a better picture of which implementations provide support for this group, we examined the default web pages of these hosts to identify companies and products, which

we show in Table 7.

Of the the 307 K HTTPS hosts that both use non-safe primes and reuse exponents, 277 K (90%) belong to hosts behind Amazon's Elastic Load Balancer [36]. These hosts use a 1024-bit prime with a 160-bit subgroup. We set up our own load balancer instance and found that the implementation failed to validate subgroup order. We were able to use a small-subgroup key recovery attack to compute 17 bits of our load balancer's private Diffie-Hellman exponent x_s in only 3813 queries. We responsibly disclosed this vulnerability to Amazon. Amazon informed us that they have removed Diffie-Hellman from their recommended ELB security policy, and are encouraging customers to use the latest policy. In May 2016, we performed additional scans and found that 88% of hosts using this prime no longer repeated exponents. We give a partial factorization for $p-1$ in Table 13; the next largest subgroups have 61 and 89 bits and an offline attack against the remaining bits of a 160-bit exponent would take 2^{71} time. For more details on the computation, see Section 2.6.

SSLeay [121], a predecessor for OpenSSL, includes several default Diffie-Hellman primes, including a 512-bit prime. We found that 717 SMTP servers used a version of the OpenSSL 512-bit prime with a single character difference in the hexadecimal representation. The resulting modulus that these servers use for their Diffie-Hellman key exchange is no longer prime. We include the factorization of this modulus along with the factors of the resulting

Company	**Product(s)**	**Count**
Ubiquiti Networks	airOS/EdgeOS	272,690
Cisco	DPC3848VM Gateway	65,026
WatchGuard	Fireware XTM	62,682
Supermicro	IPMI	42,973
ASUS	AiCloud	39,749
Electric Sheep Fencing	pfSense	14,218
Bouygues Telecom	Bbox	13,387
Other	—	135,432

Table 7: **HTTPS support for RFC5114 Group 22**—In a 100% HTTPS scan performed in October 2016, we found that of the 12,835,911 hosts that accepted Diffie-Hellman key exchange, 901,656 used Group 22. We were able to download default web pages for 646,157 of these hosts, which we examined to identify companies and products.

group order in Table 13. The use of a composite modulus further decreases the work required to perform a small subgroup attack.

Although TLS also includes static Diffie-Hellman cipher suites that require a DSS certificate, we did not include them in our study; no browser supports static Diffie-Hellman [168], and Censys shows no hosts with DSS certificates, with only 652 total hosts with non-RSA or ECDSA certificates.

2.4. IPsec

IPsec is a set of Layer-3 protocols which add confidentiality, data protection, sender authentication, and access control to IP traffic. IPsec is commonly used to implement VPNs. IPsec uses the Internet Key Exchange (IKE) protocol to determine the keys used to secure a session. IPsec may use IKEv1 [164] or IKEv2 [188]. While IKEv2 is not backwards-compatible with IKEv1, the two protocols are similar in message structure and purpose. Both versions use Diffie-Hellman to negotiate shared secrets. The groups used are limited to a fixed set of pre-determined choices, which include the DSA groups from RFC 5114, each assigned a number by IANA [188, 193, 206].

IKEv1. IKEv1 [164, 212, 258] has two basic methods for authenticated key exchange: Main Mode and Aggressive Mode. Main Mode requires six messages to establish the requisite state. The initiator sends a Security Association (`SA`) payload, containing a selection of cipher suites and Diffie-Hellman groups they are willing to negotiate. The responder selects a cipher and responds with its own `SA` payload. After the cipher suite is selected, the initiator and responder both transmit Key Exchange (`KE`) payloads containing public Diffie-Hellman values for the chosen group. At this point, both parties compute shared key materials, denoted `SKEYID`. When using signatures for authentication, `SKEYID` is computed $\texttt{SKEYID} = \text{prf}(N_i|N_r, g^{x_i x_r})$. For the other two authentication modes, pre-shared key and public-key encryption, `SKEYID` is derived from the pre-shared key and session cookies, respectively, and does not depend on the negotiated Diffie-Hellman shared secret.

Each party then in turn sends an authentication message (`AUTH`) derived from a hash over

SKEYID and the handshake. The authentication messages are encrypted and authenticated using keys derived from the Diffie-Hellman secret $g^{x_i x_r}$. The responder only sends her AUTH message after receiving and validating the initiator's AUTH message.

Aggressive Mode operates identically to Main Mode, but in order to reduce latency, the initiator sends SA and KE messages together, and the responder replies with its SA, KE, and AUTH messages together. In aggressive mode, the responder sends an authentication message first, and the authentication messages are not encrypted.

IKEv2. IKEv2 [188, 189] combines the SA and KE messages into a single message. The initiator provides a best guess ciphersuite for the KE message. If the responder accepts that proposal and chooses not to renegotiate, the responder replies with a single message containing both SA and KE payloads. Both parties then send and verify AUTH messages, starting with the initiator. The authentication messages are encrypted using session keys derived from the SKEYSEED value which is derived from the negotiated Diffie-Hellman shared secret. The standard authentication modes use public-key signatures over the handshake values.

2.4.1. Small Subgroup Attacks in IPsec

There are several variants of small subgroup attacks against IKEv1 and IKEv2. We describe the attacks against these protocols together in this section.

Small subgroup confinement attacks. First, consider attacks that can be carried out by an attacking initiator or responder. In IKEv1 Main Mode and in IKEv2, either peer can carry out a small subgroup confinement attack against the other by sending a generator of a small subgroup as its key exchange value. The attacking peer must then guess the other peer's view of the Diffie-Hellman shared secret to compute the session keys to encrypt its authentication message, leading to a mostly online attack. However, in IKEv1 Aggressive Mode, the responder sends its AUTH message before the initiator, and this value is not encrypted with a session key. If signature authentication is being used, the SKEYID and resulting hashes are derived from the Diffie-Hellman shared secret, so the initiator can

perform an offline brute-force attack against the responder's authentication message to learn their exponent in the small subgroup.

Now, consider a man-in-the-middle attacker. Bhargavan, Delignat-Lavaud, and Pironti [67] describe a transcript synchronization attack against IKEv2 that relies on a small subgroup confinement attack. A man-in-the-middle attacker initiates simultaneous connections with an initiator and a responder using identical nonces, and sends a generator g_i for a subgroup of small order q_i to each as its `KE` message. The two sides have a $1/q_i$ chance of negotiating an identical shared secret, so an authentication method depending only on nonces and shared secrets could be forwarded, and the session keys would be identical.

If the attacker also has knowledge of the secrets used for authentication, more attacks are possible. Similar to the attack described for TLS, such an attacker can use a small subgroup confinement attack to force a connection to use weak encryption. The attacker only needs to rewrite a small number of handshake messages; any further encrypted communications can then be decrypted at leisure without requiring the man-in-the-middle attacker to continuously rewrite the connection. We consider a man-in-the-middle attacker who modifies the key exchange message from both the initiator and the responder to substitute a generator g_i of a subgroup of small order q_i. The attacker must then replace the handshake authentication messages, which would require knowledge of the long-term authentication secret. We describe this attack for each of pre-shared key, signatures, and public-key authentication.

For pre-shared key authentication in IKEv1 Main Mode, IKEv1 Aggressive Mode, and IKEv2, the man-in-the-middle attacker must only know the pre-shared key to construct the authentication hash; the authentication message does not depend on the negotiated Diffie-Hellman shared secret. With probability $1/q_i$, the two parties will agree on the Diffie-Hellman shared secret. The attacker can then brute force this value after viewing messages encrypted with keys derived from it.

For signature authentication in IKEv1 Main Mode and in IKEv2, the signed hash trans-

mitted from each side is derived from the nonces and the negotiated shared secret, which is confined to one of q_i possible values. The attacker must know the private signing keys for both initiator and responder and brute force `SKEYID` from the received signature in order to forge the modified authentication signatures on each side. The communicating parties will have a q_i chance of agreeing on the same value for the shared secret to allow the attack to succeed. For IKEv1 Aggressive Mode, the attack can be made to succeed every time. The responder's key exchange message is sent together with their signature which depends on the negotiated shared secret, so the man-in-the-middle attacker can brute force the q_i possible values of the responders private key x_r and replace the responder's key exchange message with $q_i^{x_r}$, forging an appropriate signature with their knowledge of the signing key.

For public key authentication in IKEv1 Main Mode, IKEv1 Aggressive Mode, and IKEv2, the attacker must know the private keys corresponding to the public keys used to encrypt the ID and nonce values on both sides in order to forge a valid authentication hash. Since the authentication does not depend on the shared Diffie-Hellman negotiated value, a man-in-the-middle attacker must then brute force the negotiated shared key once they receives a message encrypted with the derived key. The two parties will agree on their view of the shared key with probability $1/q_i$, allowing the attack to succeed.

Small subgroup key recovery attacks. Similar to TLS, an IKE responder that reuses private exponents and does not verify that the initiator key exchange values are in the correct subgroup is vulnerable to a small subgroup key recovery attack. The most recent version of the IKEv2 specification has a section discussing reuse of Diffie-Hellman exponents, and states that "because computing Diffie-Hellman exponentials is computationally expensive, an endpoint may find it advantageous to reuse those exponentials for multiple connection setups" [188]. Following this recommendation could leave a host open to a key recovery attack, depending on how exponent reuse is implemented. A small subgroup key recovery attack on IKE would be primarily offline for IKEv1 with signature authentication and for IKEv2 against the initiator.

For each subgroup of order q_i, the attacker's goal is to obtain a responder `AUTH` message, which depends on the secret chosen by the responder. If an `AUTH` message can be obtained, the attacker can brute-force the responder's secret within the subgroup offline. This is possible if the server supports IKEv1 Aggressive Mode, since the server authenticates before the client, and signature authentication produces a value dependent on the negotiated secret. In all other IKE modes, the client authenticates first, leading to an online attack. The flow of the attack is identical to TLS; for more details see Section 2.3.

Ferguson and Schneier [126] describe a hypothetical small-subgroup attack against the initiator where a man-in-the-middle attacker abuses undefined behavior with respect to UDP packet retransmissions. A malicious party could "retransmit" many key exchange messages to an initiator and potentially receive a different authentication message in response to each, allowing a mostly offline key recovery attack.

2.4.2. Implementations

We examined several open-source IKE implementations to understand server behavior. In particular, we looked for implementations that generate small Diffie-Hellman exponents, repeat exponents across multiple connections, or do not correctly validate subgroup order. Despite the suggestion in IKEv2 RFC 7296 to reuse exponents [188], none of the implementations that we examined reused secret exponents.

All implementations we reviewed are based on FreeS/WAN [3], a reference implementation of IPSec. The final release of FreeS/Wan, version 2.06, was released in 2004. Version 2.04 was forked into Openswan [17] and strongSwan[285], with a further fork of Openswan into Libreswan [5] in 2012. The final release of FreeS/WAN used constant length 256-bit exponents but did not support RFC 5114 DSA groups, offering only the Oakley 1024-bit and 1536-bit groups that use safe primes.

Openswan does not generate keys with short exponents. By default, RFC 5114 groups are not supported, although there is a compile-time option that can be explicitly set to enable support for DSA groups. strongSwan both supports RFC 5114 groups and has explicit

			Client key exchange public values offered...		
Protocol	**Groups Offered**	**Support**	**1 mod p**	**-1 mod p**	**g_s mod p**
IKEv1	Group 22	332.4 K	82.6 K	78.5 K	332.4 K
	Group 23	333.4 K	82.5 K	82.5 K	333.4 K
	Group 24	379.8 K	93.9 K	95.2 K	379.8 K
	Baseline (Groups 2, 14, 22, 23, 24)	1139.3 K	–	–	–
IKEv2	Group 22	182.1 K	553	553	181.9 K
	Group 23	181.9 K	542	550	180.1 K
	Group 24	213.0 K	2245	2173	200.0 K
	Baseline (Groups 2, 14, 19, 20, 22, 23, 24)	1203.7 K	–	–	–

Table 8: **IKE group support and validation**—We measured support for RFC5114 DSA groups in IKEv1 and IKEv2 and test for key exchange validation by performing a series of 100% IPv4 scans in October 2016. For Group 23, g_s is a generator of a subgroup with order 3, and for Groups 22 and 24, g_s is a generator of a subgroup of order 7.

hard-coded exponent sizes for each group. The exponent size for each of the RFC 5114 DSA groups matches the subgroup size. However, these exponent sizes are only used if the `dh_exponent_ansi_x9_42` configuration option is set. It also includes a routine inside an `#ifdef` that validates subgroup order by checking that $g^q \equiv 1 \bmod p$, but validation is not enabled by default. Libreswan uses Mozilla Network Security Services (NSS) [231] to generate Diffie-Hellman keys. As discussed in Section 2.3.3, NSS generates short exponents for Diffie-Hellman groups. Libreswan was forked from Openswan after support for RFC 5114 was added, and retains support for those groups if it is configured to use them.

Although none of the implementations we examined were configured to reuse Diffie-Hellman exponents across connections, the failure to validate subgroup orders even for the pre-specified groups renders these implementations fragile to future changes and vulnerable to subgroup confinement attacks.

Several closed source implementations also provide support for RFC 5114 Group 24. These include Cisco's IOS [95], Juniper's Junos [185], and Windows Server 2012 R2 [223]. We were unable to examine the source code for these implementations to determine whether or not they validate subgroup order.

2.4.3. Measurements

We performed a series of Internet scans using ZMap to identify IKE responders. In our analysis, we only consider hosts that respond to our ZMap scan probes. Many IKE hosts

that filter their connections based on IP are excluded from our results. We further note that, depending on VPN server configurations, some responders may continue with a negotiation that uses weak parameters until they are able to identify a configuration for the connecting initiator. At that point, they might reject the connection. As an unauthenticated initiator, we have no way of distinguishing this behavior from the behaviour of a VPN server that legitimately accepts weak parameters. For a more detailed explanation of possible IKE responder behaviors in response to scanning probes, see Wouters [305].

In October 2016, we performed a series of scans offering the most common cipher suites and group parameters we found in implementations to establish a baseline population for IKEv1 and IKEv2 responses. For IKEv1, the baseline scan offered Oakley groups 2 and 14 and RFC 5114 groups 22, 23, and 24 for the group parameters; SHA1 or SHA256 for the hash function; pre-shared key or RSA signatures for the authentication method; and AES-CBC, 3DES, and DES for the encryption algorithm. Our IKEv2 baseline scan was similar, but also offered the 256-bit and 384-bit ECP groups and AES-GCM for authenticated encryption.

On top of the baseline scans, we performed additional scans to measure support for the non-safe RFC 5114 groups and for key exchange parameter validation. Table 8 shows the results of the October IKE scans. For each RFC 5114 DSA group, we performed four handshakes with each host; the first tested for support by sending a valid client key exchange value, and the three others tested values that should be rejected by a properly-validating host. We did not scan using the key exchange value 0 because of a vulnerability present in unpatched Libreswan and Openswan implementations that causes the IKE daemon to restart when it receives such a value [26].

We considered a host to accept our key exchange value if after receiving the value, it continued the handshake without any indication of an error. We found that 33.2% of IKEv1 hosts and 17.7% of IKEv2 hosts that responded to our baseline scans supported using one of the RFC 5114 groups, and that a surprising number of hosts failed to validate key exchange values. 24.8% of IKEv1 hosts that accepted Group 23 with a valid key exchange value

also accepted 1 mod p or -1 mod p as a key exchange value, even though this is explicitly warned against in the RFC [249]. This behavior leaves these hosts open to a small subgroup confinement attack even for safe primes, as described in Section 2.2.6.

For safe groups, a check that the key exchange value is strictly between 1 and $p - 1$ is sufficient validation. However, when using non-safe DSA primes, it is also necessary to verify that the key exchange value lies within the correct subgroup (i.e., $y^q \equiv 1 \bmod p$). To test this case, we constructed a generator of a subgroup that was not the intended DSA subgroup, and offered that as our key exchange value. We did not find any IKEv1 hosts that rejected this key exchange value after previously accepting a valid key exchange value for the given group. For IKEv2, the results were similar with the exception of Group 24, where still over 93% of hosts accepted this key exchange value. This suggests that almost no hosts supporting DSA groups are correctly validating subgroup order.

We observed that across all of the IKE scans, 109 IKEv1 hosts and 52 IKEv2 hosts repeated a key exchange value. This may be due to entropy issues in key generation rather than static Diffie-Hellman exponents; we also found 15,891 repeated key exchange values across different IP addresses. We found no hosts that used both repeated key exchange values and non-safe groups. We summarize these results in Table 4.

2.5. SSH

SSH contains three key agreement methods that make use of Diffie-Hellman. The "Group 1" and "Group 14" methods denote Oakley Group 2 and Oakley Group 14, respectively [311]. Both of these groups use safe primes. The third method, "Group Exchange", allows server to select a custom group [131]. The group exchange RFC specifies that all custom groups should use safe primes. Despite this, RFC 5114 notes that group exchange method allows for its DSA groups in SSH, and advocates for their immediate inclusion [206].

In all Diffie-Hellman key agreement methods, after negotiating cipher selection and group parameters, the SSH client generates a public key exchange value $y_c = g^{x_c} \bmod p$ and sends it to the server. The server computes its own Diffie-Hellman public value $y_s = g^{x_s} \bmod p$

and sends it to the client, along with a signature from its host key over the resulting shared secret $Y = g^{x_s x_c} \bmod p$ and the hash of the handshake so far. The client verifies the signature before continuing.

2.5.1. Small Subgroup Attacks in SSH

Small subgroup confinement attacks. An SSH client could execute a small subgroup confinement attack against an SSH server by sending a generator g_i for a subgroup of small order q_i as its client key exchange, and immediately receive the server's key exchange $g^{x_s} \bmod p$ together with a signature that depends on the server's view of the shared secret $Y_s = g_i^{x_s} \bmod p$. For small q_i, this allows the client to brute force the value of $x_s \bmod q_i$ offline and compare to the server's signed handshake to learn the correct value of $x_s \bmod q_i$. To avoid this, the SSH RFC specifically recommends using safe primes, and to use exponents at least twice the length of key material derived from the shared secret [131].

If client and server support Diffie-Hellman group exchange and the server uses a non-safe prime, a man in the middle with knowledge of the server's long-term private signing key can use a small subgroup confinement attack to man-in-the-middle the connection without having to rewrite every message. The attack is similar to the case of TLS: the man in the middle modifies the server group and key exchange messages, leaving the prime unchanged, but substituting a generator g_i of a subgroup of small order q_i for the group generator and g_i for the server's key exchange value y_s. The client then responds with a client key exchange message $y_c = g_i^{x_c} \bmod p$, which the man in the middle leaves unchanged. The attacker then forges a correct signature for the modified server group and key exchange messages and passes it to the client. The server's view of the shared secret is $g_i^{x_c x_s} \bmod p$, and the client's view of the shared secret is $g_i^{x_c} \bmod p$. As in the attack described for TLS, these views are identical when $x_s \equiv 1 \bmod q_i$, so this connection will succeed with probability $1/q_i$. For a small enough q_i, this enables a man in the middle to use a compromised server signing key to decrypt traffic with a reasonable probability of success, while only requiring tampering with the initial handshake messages, rather than having to actively rewrite the entire connection for the duration of the session.

Small subgroup key recovery attacks. Since the server immediately sends a signature over the public values and the Diffie-Hellman shared secret, an implementation using static exponents and non-safe primes that is vulnerable to a small subgroup confinement attack would also be vulnerable to a mostly offline key recovery attack, as a malicious client would only need to send a single key exchange message per subgroup.

2.5.2. Implementations

Censys [2] SSH banner scans show that the two most common SSH server implementations are Dropbear and OpenSSH. Dropbear group exchange uses hard-coded safe prime parameters from the Oakley groups and validates that client key exchange values are greater than 1 and less than $p-1$. While OpenSSH only includes safe primes by default, it does provide the ability to add additional primes and does not provide the ability to specify subgroup orders. Both OpenSSH and Dropbear generate fresh exponents per connection.

We find one SSH implementation, Cerberus SFTP server (FTP over SSH), repeating server exponents across connections. Cerberus uses OpenSSL, but fails to set `SSL_OP_SINGLE_DH_USE`, which was required to avoid exponent reuse prior to OpenSSL 1.0.2f.

2.5.3. Measurements

Of the 15.2 M SSH servers on Censys, of which 10.7 M support Diffie-Hellman group exchange, we found that 281 used a non-safe prime, and that 1.1 K reused Diffie-Hellman exponents. All but 26 of the hosts that reused exponents had banners identifying the Cerberus SFTP server. We encountered no servers that both reused exponents and used non-safe primes.

Key Exchange Value	**Handshake Initiated**	**Accepted**
$0 \bmod p$	175.6 K	5.7 K
$1 \bmod p$	175.0 K	43.9 K
$-1 \bmod p$	176.0 K	59.0 K

Table 9: **SSH validation**—In a 1% SSH scan performed in February 2016, we sent the key exchange values $y_c = 0, 1$ and $p-1$. We count hosts as having initiated a handshake if they send a `SSH_MSG_KEX_DH_GEX_GROUP`, and we count hosts as "Accepted" if they reply to the client key exchange message with a `SSH_MSG_KEX_DH_GEX_REPLY`.

Prime	Exact Order Known							Exact Order Unknown	
lg(p)	160 bits	224 bits	256 bits	300 bits	lg(p) − 8	lg(p) − 32	lg(p) − 64	Unlikely DSA	Likely DSA
512	3	0	0	0	5	0	0	760	43
768	4	0	0	4	2,685	0	0	220	1,402
1024	29	0	0	0	323	944	176	1,559	26,881
2048	0	1	1	0	0	0	0	1,128	4,890
3072	0	0	0	0	0	5	0	9	152
4096	4	0	0	0	0	0	0	20	183
8192	0	0	0	0	0	0	0	0	1
Other	0	0	0	0	0	0	0	400	15

Table 10: **Distribution of orders for groups with non-safe primes**—For groups for which we were able to determine the subgroup order exactly, 160-bits subgroup orders are common. We classify other groups to be likely DSA groups if we know that the subgroup order is at least 8 bits smaller than the prime.

We performed a scan of 1% of SSH hosts in February 2016 offering the key exchange values of $y_c = 0 \bmod p, 1 \bmod p$ and $p - 1 \bmod p$. As Table 9 shows, 33% of SSH hosts failed to validate group order when we sent the key exchange value $p - 1 \bmod p$. Even when safe groups are used, this behaviour allows an attacker to learn a single bit of the private exponent, violating the decisional Diffie-Hellman assumption and leaving the implementation open to a small subgroup confinement attack (Section 2.3.1).

2.6. Factoring Group Orders of Non-Safe Primes

Across all scans, we collected 41,847 unique groups with non-safe primes. To measure the extent to which each group would facilitate a small subgroup attack in a vulnerable implementation, we attempted to factor $(p-1)/2$. We used the GMP-ECM [150] implementation of the elliptic curve method for integer factorization on a local cluster with 288 cores over a several-week period to opportunistically find small factors of the group order for each of the primes.

Given a group with prime p and a generator g, we can check whether the generator generates the entire group or generates a subgroup by testing whether $g^{q_i} \equiv 1 \bmod p$ for each factor q_i of $(p-1)/2$. When $g^{q_i} \equiv 1 \bmod p$, then if q_i is prime, we know that q_i is the exact order of the subgroup generated by g; otherwise q_i is a multiple of the order of the subgroup. We show the distribution of group order for groups using non-safe primes in Table 10. We were able to completely factor $p-1$ for 4,701 primes. For the remaining primes, we did not obtain enough factors of $(p-1)/2$ to determine the group order.

Of the groups where we were able to deduce the exact subgroup orders, several thousand had a generator for a subgroup that was either 8, 32, or 64 bits shorter than the prime itself. Most of these were generated by the Xlight FTP server, a closed-source implementation supporting SFTP. It is not clear whether this behavior is intentional or a bug in an implementation intending to generate safe primes. Primes of this form would lead to a more limited subgroup confinement or key recovery attack.

Given the factorization of $(p-1)/2$, and a limit for the amount of online and offline work an attacker is willing to invest, we can estimate the vulnerability of a given group to a hypothetical small subgroup key recovery attack. For each subgroup of order q_i, where q_i is less than the online work limit, we can learn q_i bits of the secret key via an online brute-force attack over all elements of the subgroup. To recover the remaining bits of the secret key, an attacker could use the Pollard lambda algorithm, which runs in time proportional to the square root of the remaining search space. If this runtime is less than the offline work limit, we can recover the entire secret key. We give work estimates for the primes we were able to factor and the number of hosts that would be affected by such a hypothetical attack in Table 11.

The DSA groups introduced in RFC 5114 [206] are of particular interest. We were able to completely factor $(p-1)/2$ for both Group 22 and Group 24, and found several factors for Group 23. We give these factorizations in Table 13. In Table 12, we show the amount of online and offline work required to recover a secret exponent for each of the RFC 5114 groups. In particular, an exponent of the recommended size used with Group 23 is fully recoverable via a small subgroup attack with 33 bits of online work and 47 bits of offline work.

2.7. Discussion

Small subgroup attacks require a number of special conditions to go wrong in order to be feasible. For the case of small subgroup confinement attacks, a server must both use a non-safe group and fail to validate subgroup order; the widespread failure of implementations

	Work (bits)		*HTTPS*		*MAIL*		*SSH*	
Exponent	**Online**	**Offline**	**Groups**	**Hosts**	**Groups**	**Hosts**	**Groups**	**Hosts**
160	20	30	3	2	3	7	0	0
160	30	45	517	1,996	1963	1,143,524	11	10
160	40	60	3,701	8,495	13,547	1,159,853	109	68
224	20	30	0	0	0	0	0	0
224	30	45	2	2	14	16	0	0
224	40	60	307	691	1039	1,141,840	3	1
256	20	30	0	0	0	0	0	0
256	30	45	0	0	1	1	0	0
256	40	60	42	478	180	1,140,668	0	0

Table 11: **Full key recovery attack complexity**—We estimate the amount of work required to carry out a small subgroup key recovery attack, and show the prevalence of those groups in the wild. Hosts are vulnerable if they reuse exponents and fail to check subgroup order.

to implement or enable group order validation means that large numbers of hosts using non-"safe" primes are vulnerable to this type of attack.

For a full key recovery attack to be possible the server must additionally reuse a small static exponent. In one sense, it is surprising that any implementations might satisfy all of the requirements for a full key recovery attack at once. However, when considering all of the choices that cryptographic libraries leave to application developers when using Diffie-Hellman, it is surprising that any protocol implementations manage to use Diffie-Hellman securely at all.

We now use our results to draw lessons for the security and cryptographic communities, provide recommendations for future cryptographic protocols, and suggest further research.

Group	**Exponent Size**	**Online Work**	**Offline Work**
Group 22	160	8	72
Group 23	224	33	47
Group 24	256	32	94

Table 12: **Attacking RFC 5114 groups**—We show the log of the amount of work in bits required to perform a small subgroup key recovery attack against a server that both uses a static Diffie-Hellman exponent of the same size as the subgroup order and fails to check group order.

Source	Factored Completely?	Order Factorization
RFC 5114 Group 22	Yes	2^3 * 7 * df * 183a872bdc5f7a7e88170937189 * 228c5a311384c02e1f287c6b7b2d * 5a857d66c65a60728c353e32ece8be1 * f518aa8781a8df278aba4e7d64b7cb9d49462353 * 1a3adf8d6a69682661ca6e590b447e66ebd1bbdeab5e6f3744f06f46cf2a8300622ed50011479f18143d471a53d30113995663a447dcb8e81bc24d988edc41f21
RFC 5114 Group 23	No	3^2 * 5 * 2b * 49 * 9d * 5e9a5 * 93ee1 * 2c3f0539 * 136c58359 * 1a30b7358d * 335a378eb0d * 801c0d34c58d93fe997177101f80535a4738cebcbf389a99b36371eb * 22bbe4b573f6fc6dc24fef3f56e1c216523b3210d27b6c078b32b842aa48d35f230324e48f6dc2a10dd23d28d382843a78f264495542be4a95cb05e41f80b013f8b0e3ea26b84cd497b43cc932638530a068ecc44af8ea3cc84139f0667100d426b60b9ab82b8de865b0cbd633f41366622011006632e0832e827febb7066efe4ab4f1b2e99d96adfaf1721447b167cb49c372efcb82923b3731433cecb7ec3ebbc8d67ef441b5d11fb3328851084f74de823b5402f6b038172348a147b1ceac47722e31a72fe68b44ef4b
RFC 5114 Group 24	Yes	7 * d * 9f5 * 22acf * bd9f34b1 * 8cf83642a709a097b447997640129da299b1a47d1eb3750ba308b0fe64f5fbd3 * 15adfe949ebb242e5cd0978fac1b43fdbd2e5b0c5f48924fbbd370195c0eb20596d98ad0a9e3fd98876413d926f41a8b918d2ec4b018a30efe5e336bf3c7ce60d515cf46af5facf3bb389f68ad0c4ed2f0b1dbb970293741eb6509c64e731802259a639a7f57d4a9c0d9445241f5bcdbdc50555b76d9c335c1fa4e11a8351f1bf4730dd67ffed877cc13e8ea40c7d51441c1f4e59155ef1159eca75a2359f5e0284cd7f3b982c32e5c51dbf51b45f4603ef46bae528739315ca679703c1ffcf3b44fe3da5999daadf5606eb828fc57e46561be8c6a866361
Amazon Load Balancer	No	2 * 3 * 5 * edb * 181ac5dbfe5ce13b * 18aa349859e9e9de09b7d65 * 9414a18a7b575e8f42f6cb2dbc22eb1fc21d4929 * 2de9f1171a2493d46a31d508b63532cdf86d21db6f50f717736fc4b0b722856a504ed4916e0484fe4ba5f5f4a9fff28a1233b728b3d043aec37c4f138ffd58fe7a8c3c1e93cb52be527395e45db487b61daadded9c8ec35
Mistyped OpenSSL 512 "Prime" Factors	Yes	5 * b * a9b461e1636f4b51ef * 1851583cf5f9f731364e4aa6cdc2cac4f01* 3f0b39cacfc086df4baf46c7fa7d1f4dfe184f9d22848325a91c519f79023a4526d8369e86b
Mistyped OpenSSL 512 Order Factors	Yes	2^13 * 3^3 * 5^2 * 11^2 * 269 * 295 * 4d5 * 97c3 * 9acfe7 * 8cdd0e128f * 385b564eecd613536818f949 * 146d410923e999f8c291048dc6feffcebf8b9e99eec9a4d585f87422e49b393256c23c9

Table 13: **Group order factorization for common non-safe primes**—We used the elliptic curve method to factor $(p-1)/2$ for each of the non-safe primes we found while scanning, as well as the mistyped OpenSSL "prime".

RFC 5114 design rationale. Neither NIST SP 800-56A nor RFC 5114 give a technical justification for fixing a much smaller subgroup order than the prime size. Using a shorter private exponent comes with performance benefits. However, there are no known attacks that would render a short exponent used with a safe prime less secure than an equivalently-sized exponent used with in a subgroup with order matched to the exponent length. The cryptanalysis of both short exponents and small subgroups are decades old.

If anything, the need to perform an additional modular exponentiation to validate subgroup order makes Diffie-Hellman over DSA groups *more* expensive than the safe prime case, for identical exponent lengths. As a more minor effect, a number field sieve-based cryptanalytic attack against a DSA prime is computationally slightly easier than against a safe prime. The design rationale may have its roots in preferring to implicitly use the assumption that Diffie-Hellman is secure for a small prime-order subgroup without conditions on exponent length, rather than assuming Diffie-Hellman with short exponents is secure inside a group of much larger order. Alternatively, this insistence may stem from the fact that the security of DSA digital signatures requires the secret exponent to be uniformly random, although

no such analogous attacks are known for Diffie-Hellman key exchange [234]. Unfortunately, our empirical results show that the necessity to specify and validate subgroup order for Diffie-Hellman key exchange makes implementations more fragile in practice.

Cryptographic API design. Most cryptographic libraries are designed with a large number of potential options and knobs to be tuned, leaving too many security-critical choices to the developers, who may struggle to remain current with the diverse and ever-increasing array of cryptographic attacks. These exposed knobs are likely due to a prioritization of performance over security. These confusing options in cryptographic implementations are not confined to primitive design: Georgiev et al. [146] discovered that SSL certificate validation was broken in a large number of non-browser TLS applications due to developers misunderstanding and misusing library calls. In the case of the small subgroup attacks, activating most of the conditions required for the attack will provide slight performance gains for an application: using a small exponent decreases the work required for exponentiation, reusing Diffie-Hellman exponents saves time in key generation, and failing to validate subgroup order saves another exponentiation. It is not reasonable to assume that applications developers have enough understanding of algebraic groups to be able to make the appropriate choices to optimize performance while still providing sufficient security for their implementation.

Cryptographic standards. Cryptographic recommendations from standards committees are often too weak or vague, and, if strayed from, provide little recourse. The purpose of standardized groups and standardized validation procedures is to help remove the onus from application developers to know and understand the details of the cryptographic attacks. A developer should not have to understand the inner workings of Pollard lambda and the number field sieve in order to size an exponent; this should be clearly and unambiguously defined in a standard. However, the tangle of RFCs and standards attempting to define current best practices in key generation and parameter sizing do not paint a clear picture, and instead describe complex combinations of approaches and parameters, exposing the

fragility of the cryptographic ecosystem. As a result, developers often forget or ignore edge cases, leaving many implementations of Diffie-Hellman too close to vulnerable for comfort. Rather than provide the bare minimums for security, the cryptographic recommendations from standards bodies should be designed for defense-in-depth such that a single mistake on the part of a developer does not have disastrous consequences for security. The principle of defense-in-depth has been a staple of the systems security community; cryptographic standards should similarly be designed to avoid fragility.

Protocol design. The interactions between cryptographic primitives and the needs of protocol designs can be complex. The after-the-fact introduction of RFC 5114 primes illustrates some of the unexpected difficulties: both IKE and SSH specified group validation only for safe primes, and a further RFC specifying extra group validation checks needed to be defined for IKE. Designing protocols to encompass many unnecessary functions, options, and extensions leaves room for implementation errors and makes security analysis burdensome. IKE is a notorious example of a difficult-to-implement protocol with many edge cases. Just Fast Keying (JFK), a protocol created as a successor to IKEv1, was designed to be an exceedingly simple key exchange protocol without the unnecessarily complicated negotiations present in IKE [30]. However, the IETF instead standardized IKEv2, which is nearly as complicated as IKEv1. Protocols and cryptosystems should be designed with the developer in mind—easy to implement and verify, with limited edge cases. The worst possible outcome is a system that appears to work, but provides less security than expected.

To construct such cryptosystems, secure-by-default primitives are key. As we show in this paper, finite-field based Diffie-Hellman has many edge cases that make its correct use difficult, and which occasionally arise as bugs at the protocol level. For example, SSH and TLS allow the server to generate arbitrary group parameters and send them to the client, but provide no mechanism for the server to specify the group order so that the client can validate the parameters. Diffie-Hellman key exchange over groups with different properties cannot be treated as a black-box primitive at the protocol level.

Recommendations. As a concrete recommendation, modern Diffie-Hellman implementations should prefer elliptic curve groups over safe curves with proper point validation [58]. These groups are much more efficient and have shorter key sizes than finite-field Diffie-Hellman at equivalent security levels. The TLSv1.3 standard includes a list of named curves designed to modern security standards [107]. If elliptic curve Diffie-Hellman is not an option, then implementations should follow the guidelines outlined in RFC 7919 for selecting finite field Diffie-Hellman primes [148]. Specifically, implementations should prefer "safe" primes of documented provenance of at least 2048 bits, validate that key exchange values are strictly between 1 and $p - 1$, use ephemeral key exchange values for every connection, and use exponents of at least 224 bits.

CHAPTER 3 : Measuring elliptic curve Diffie-Hellman

3.1. Introduction

In 2015, Nick Sullivan outlined a theoretical parameter downgrade attack against TLS versions 1.0–1.2 which he named CurveSwap [289]. The main observation behind CurveSwap is that in the TLS handshake, the client's list of supported elliptic curves is not authenticated until the client finished message, and is authenticated only by the negotiated Diffie-Hellman secret. Thus if a man-in-the-middle attacker were able to precompute or solve an elliptic curve discrete log online for some curve, they could downgrade the connection to use that weak curve, allowing them to decrypt or modify the encrypted communications. The attack was inspired by the FREAK [61] and Logjam [29] cipher suite downgrade attacks against TLS.

In his 31C3 presentation, Sullivan concluded that the weakest commonly supported curve was sect163k, supported by 4.3% of sampled clients and 0.13% of the Alexa top 100,000 web sites. Since a 160-bit elliptic curve discrete log has yet to be publicly demonstrated, let alone computed within a TLS handshake timeout, the attack appeared to remain theoretical.

In this paper, we evaluate the feasibility of a practical CurveSwap attack by exploring the protocol-level and implementation-level attack surface of elliptic curve usage in TLS, IPsec, SSH, and JSON Web Encryption (JWE). There are a number of potential vulnerabilities in elliptic curve implementations that taken in combination could enable a CurveSwap attack, including support for curves of small order, point validation failures, and twist insecurity. We performed extensive passive and active measurements of these behaviors and implementation choices among clients and servers. Among our scans, we found populations of servers that accept invalid curve points years after flaws have been publicly disclosed and patched in common libraries, little vulnerability to twist attacks, and significant populations of hosts that repeat public key exchange values both across IP addresses and across multiple scans. However, these behaviors were not present in combinations that would lead to an

effective attack for vulnerable curves. Ultimately we conclude that TLS, IPsec, and SSH do not appear to be vulnerable on any significant scale to a feasible CurveSwap attack based on the vectors we evaluated.

Some protocol designs are much more resistant to CurveSwap-style downgrade attacks than others. We observe that the design of SSH and TLSv1.3, where the server uses their long-term authentication key to sign the entire handshake, are much more resistant to parameter downgrade attacks like CurveSwap than earlier versions of TLS.

Our survey of elliptic curve support for TLS, IPsec, and SSH gives a snapshot of elliptic curve deployments in 2017. The NIST-standardized curve secp256r1 is the most widely supported curve in our measurements, while support for other curves in our data was in general lower, with a long tail of more unusual standardized curves. Curve support varied wildly by protocol. We found small but nontrivial support for a number of 160-bit curves that only offer 80 bits of security, although only a negligible number of clients or servers preferred these curves over stronger curves. We were surprised to discover that very few hosts supported secp224r1 on any protocol, many hosts failed to respect a client's selection of elliptic curves, and that essentially no TLS hosts servers supported custom curves.

We also extensively examined source code, and discovered several vulnerabilities. The JWE protocol standard fails to mention that implementations need to perform curve validity checks, and we discovered a number of JWE libraries that were vulnerable to a classic invalid curve attack allowing an attacker to recover the private key, including Cisco's node-jose, jose2go, Nimbus JOSE+JWT and jose4j. We also discovered flaws in NSS and Java's scalar point multiplication routines that could cause them to output incorrect results given certain inputs, although these flaws do not appear to be exploitable.

3.1.1. Our Contributions

In this paper, we perform a broad survey of elliptic curve cryptography on the public Internet. The maze of different standards, curves, and implementation choices for elliptic curve cryptography makes a holistic evaluation of our cryptographic infrastructure quite

challenging. We measure the landscape of elliptic curve implementations on the Internet with passive and active measurements, describe known and new attack vectors against ECC, and examine source code to find implementation vulnerabilities.

- **Active Measurements** We perform Internet-wide scans of TLS, SSH, and IPsec servers to measure elliptic curve support and implementation behaviors.
- **Passive Measurements** We measure TLS client support and preferences for elliptic curves.
- **Protocol Analysis** We explore analogues of CurveSwap for IPsec and SSH. We also survey attacks against elliptic curves and evaluate their impact on the CurveSwap attack for TLS.
- **Source Code Analysis** We extensively examined source code and found widespread invalid curve vulnerabilities in JWE libraries, as well as flawed scalar multiplication routines in Java and NSS.

Although some elliptic curve implementations have fallen victim to known implementation pitfalls, for TLS, SSH, and IPsec, most hosts appear to resist known attacks. We conclude that protocol designers should continue to build in defense in depth.

3.1.2. Disclosure and Mitigations

In February 2017 we submitted bug reports to the developers of several libraries implementing JSON Web Encryption (JWE, RFC 7516) that were vulnerable to invalid curve attacks, including Cisco's node-jose, jose2go, Nimbus JOSE+JWT and jose4j. They have all acknowledged the issue and released a patch. We also described the nature of the invalid curve attack applied to JWE in a blog post [274]. We reported the NSS vulnerability to Mozilla in March 2017. NSS fixed the issue in the 3.31 release. We reported the Java vulnerability to Oracle in March 2017. Oracle issued a patch that fixes the issue on July 18, 2017. We also disclosed these vulnerabilities to the public in a blog post [275].

3.2. Preliminaries

Elliptic curve cryptography can be used for key exchange, asymmetric encryption, or for signatures. Among widely implemented public key primitives, elliptic curves offer the best resistance to cryptanalytic attacks on classical computers, and as a result can be used with smaller key sizes than RSA or finite field based discrete logarithm schemes. In this paper, we focus on elliptic curve Diffie-Hellman key exchange.

3.2.1. Elliptic Curve Cryptography

A number of standards exist defining elliptic curves for use in cryptography. In 2000, the Certicom SECG published the SEC 2 specification [80] giving parameters for 33 elliptic curves of varying sizes and properties. Several of these curves were later standardized by NIST, ISO, and ANSI under different names. Other proposals for curves include the Oakley elliptic curve groups [249], the Brainpool curves [219], and more recent constructions such as Curve25519 [55], Curve41417 [60], and Curve448 [161].

Prime curves. An elliptic curve $E(\mathbb{F}_p)$ over a prime finite field $\mathbb{F}_p$ with $p \neq 2$ is the set of points $P = (x, y) \in \mathbb{F}_p^2$ that are solutions to some equation E over $\mathbb{F}_p$, together with an extra point $\mathcal{O}$, the point at infinity. It is possible to define an addition law, so that these points form a group.

Such curves are often specified in Weierstrass form $E : y^2 = x^3 + ax + b \pmod{p}$ where $a, b \in \mathbb{F}_p$ are domain parameters that define the curve. Every elliptic curve over a finite field $\mathbb{F}_p$ of a prime order can be converted to this form. Some widely-used examples of prime curves are the NIST curves from FIPS 186-4 [239] and the Brainpool curves [219].

Cryptographic applications typically work within a cyclic subgroup of prime order n. This group will be generated by a base point $G \in E(\mathbb{F}_p)$.

One can compute an element kG of this group using a scalar-by-point multiplication algorithm. The underlying hardness assumption in most elliptic curve cryptography is the elliptic curve discrete logarithm problem: given an elliptic curve $E(\mathbb{F}_p)$, a generator G, and

a point P it is hard to find a k satisfying $P = kG$. The best known algorithms for solving the elliptic curve discrete logarithm problem run in square root time in the order of the subgroup generated by the elliptic curve's generator.

Binary curves. Elliptic curves over characteristic 2 finite fields $\mathbb{F}_{2^m}$ are specified as the set of points $P = (x, y) \in \mathbb{F}_{2^m}^2$ that are solutions to the equation $E : y^2 + xy = x^3 + ax^2 + b$ in $\mathbb{F}_{2^m}$.

Recent progress on the elliptic curve discrete logarithm problem for small-characteristic fields has raised concern about the security of binary curves, although there are not yet any subexponential time attacks against curves standardized for use in the network protocols we study in this paper [134, 278].

The SEC 2 standard [80] includes parameters for a number of binary curves. The Oakley elliptic curve groups [249] are also binary curves.

Domain parameters. An elliptic curve group is defined by a set of domain parameters which consist of the following values: q, an integer that defines the order of the finite field $\mathbb{F}_q$ of the curve; a and b, the coefficients of the curve equation; G, a generator of a subgroup of prime order on the curve; n, the order of the subgroup that G generates; and h, the cofactor, which is equal to the number of curve points w divided by n.

3.2.2. ECDH Key Exchange

In this paper, we are primarily interested in elliptic curve Diffie-Hellman key exchange. To negotiate a shared secret using ECDH, Alice generates a random private key k_a, generates her public value $Q_a = k_aG$, and sends Q_a to Bob. Bob generates a random private key k_b, generates his public value $Q_b = k_bG$, and sends Q_b to Alice. Alice can then compute the shared secret as $P = k_aQ_b$ and Bob can compute it as $P = k_bQ_a$. Real-world protocols then use P to derive symmetric keys that Alice and Bob use to establish an authenticated and encrypted communication channel.

3.2.3. Scalar-by-point multiplication algorithms

The most important operation on elliptic curves for the cryptographic algorithms we study in this paper is scalar-by-point multiplication. That is, given a point P on an elliptic curve and an integer k, compute the curve point kP.

Point representation. Elliptic curve points can be represented in many different forms. The canonical representation uses *affine coordinates*, where a point on the curve is represented by a pair of integers (x, y) that satisfy the curve equation. This is called *uncompressed* point format. However, this representation requires an expensive field inversion operation to add two elliptic curve points.

Most applications of elliptic curves use only the x-coordinate of a point. A valid x-coordinate could correspond to two possible y coordinates of points on the curve, the point (x, y) or the point $(x, -y)$; these can be recovered from x using the curve equation. Thus a point can be uniquely represented by sending only the x-coordinate and the sign of the y-coordinate; this is called *compressed* format.

Double and add. The simplest algorithm to compute scalar-by-point multiplication is double-and-add. This algorithm iteratively applies the group addition law and a doubling procedure. There are a number of variants of this algorithm, such as sliding windows. However, this algorithm has the drawback that it is not secure against side channel attacks. It also requires both the x and y coordinates of the input points.

Montgomery ladder. Some elliptic curves can also be specified in Montgomery form [227]: $E : By^2 = x^3 + Ax^2 + x$. An advantage of this form is that it allows a very fast algorithm for scalar-by-point multiplication using only the x coordinate, the Montgomery ladder.

The single-coordinate version of the Montgomery ladder algorithm for scalar-by-point multiplication requires fewer arithmetic operations than standard Weierstrass scalar-by-point multiplication methods and offers better side channel resistance [184, 243]. Curve25519, in-

troduced by [55], is specified in Montgomery form, as are Curve41417 [60] and Curve448 [161] (the Goldilocks curve).

Brier-Joye. It is possible to compute an x-coordinate only scalar multiplication for Weierstrass-form elliptic curves using the Brier-Joye ladder [78]. This algorithm is constant time and has good side channel resistance. Unfortunately, it is slow.

3.2.4. Invalid Point Attacks

For most curves, ECDH implementations must validate that the public key exchange messages they receive are valid points on the correct elliptic curve, otherwise they may be vulnerable to a variety of attacks.

Small subgroup attacks. Small subgroup attacks against prime-field Diffie-Hellman were described by Lim and Lee [207]. In this type of attack, the cryptographic domain parameters specify a subgroup within a larger group. If the cofactor of the order of the correct subgroup has small prime factors p_i, an adversary could send a key exchange that lies in a subgroup of order p_i instead of the correct subgroup and use the victim's response to deduce the victim's secret modulo p_i. The attacker can then repeat this attack for different primes and use the Chinese remainder theorem to reconstruct the victim's secret modulo the product of these primes.

Elliptic curves that are standardized for cryptographic use are typically chosen to have small cofactors to limit the number of elements of small order on the curve and to limit the checks required to protect against these small subgroup attacks [55]. NIST recommends a maximum cofactor for various curve sizes [239]. The NIST curves specified in FIPS 186-4 have cofactor 1, 2, or 4. Curves in Montgomery form always have a cofactor that is a multiple of 4 [227].

One can also protect against this type of attack by checking that a received point P has the correct group order by checking that $nP = \mathcal{O}$. Alternatively, one can use ECDH with cofactor multiplication, in which both parties multiply their Diffie-Hellman shared secret

by h [79].

Invalid curve attacks. A double-and-add-based implementation of scalar multiplication that does not validate key exchange values is vulnerable to a much more severe invalid curve attack. In an invalid curve attack, the attacker sends an elliptic curve point of small order that lies on a *different curve*. This attack is due to Antipa et al. [39].

In a Weierstrass-form curve, textbook double-and-add algorithms are independent of the curve parameter b, so an attacker can search for values b' such that a curve $E' : y^2 = x^3 + ax + b'$ has points $P_i = (x_i, y_i)$ of small order q_i and send them to the victim. If the victim does not verify that the received key exchange value and computed shared secret are on the correct curve and has the correct order, the victim's response may allow the attacker to compute the victim's secret key modulo q_i.

In contrast to the Lim-Lee attack for prime-field Diffie-Hellman where an attacker is limited to the prime factors of the cofactor of the correct subgroup, the attacker in this elliptic curve scenario has much more leeway in choosing curves that have points of suitably small coprime order.

This attack can be prevented if an implementation validates that the points it receives lie on the correct curve. This attack is also somewhat mitigated by scalar-by-point multiplication algorithms that use only the x-coordinate, although these may be vulnerable to twist attacks, described below.

Curve twist attacks. A Weierstrass curve of the form $E : y^2 = x^3 + ax + b \bmod p$ is related to a twisted curve, $E' : dy^2 = x^3 + ax + b$.

Any x-coordinate has an associated pair of y coordinates that are either on the original curve or some twisted curve. If d is a quadratic residue, i.e., if there is a w with $w^2 = d \bmod p$, then E and E' are isomorphic mod p and thus have equivalent security. If d is a quadratic non-residue, E' is not isomorphic to E and the curve orders satisfy $|E| + |E'| = p + 2$. This

is called a *nontrivial quadratic twist*.

An implementation that uses a single-coordinate ladder such as the Montgomery ladder might be vulnerable to a form of invalid curve attacks in which the attacker sends an x-coordinate that lies on a weak twist of the correct curve. This type of attack is due to Foque, Lercier, Réal, and Valette [128].

The NIST-standardized curves secp192r1 and secp224r1 have weak twists that reduce the cost of such an attack to 2^{48} and 2^{59}, respectively[58, 128]. The binary curves ec2n_155 and ec2n_185 also have weak twists which reduce the attack cost to 2^{33} and 2^{47}, respectively. secp256r1 and secp384r1 have secure twists. Recent curve constructions such as Curve25519 were explicitly designed to have strong twists and not require an additional validation step. Otherwise, implementations must verify that the received coordinate lies on the correct curve.

Curve downgrade attacks. In a curve downgrade attack, a man-in-the-middle adversary interferes with a connection to cause the communicating parties to choose a weaker curve than they would otherwise negotiate. In Section 3.5, we present the CurveSwap attack against TLS, and study the feasibility of similar curve downgrade attacks against SSH and IPsec.

3.2.5. ECC in TLS

Elliptic curve use in TLS versions 1.2 and earlier is specified by RFC 4492 [72]. Elliptic curves can be used in static elliptic curve Diffie-Hellman (ECDH) and ephemeral ECDH (ECDHE) key exchange, and ECDSA signatures. In this paper, we focus on ECDHE key exchange.

Clients declare support for elliptic curves by including ECHD(E) cipher suites in their list of supported cipher suites and via the supported elliptic curves and the supported points format extensions in the client hello message. This message consists of a list of supported elliptic curves sorted by client preference, and a list of the point formats that the client can

parse. The list of supported elliptic curves can include 25 of the named curves specified in SEC 2 [80], and can also indicate support for arbitrary explicit prime or binary curves.

If the server chooses an ECDHE cipher suite, the server key exchange message includes an indication of the server's chosen curve (either named or a set of parameters for an explicit curve), the server's public key exchange value given as the encoding type and a byte string representing an elliptic curve point, and a digital signature on these two values using the server's certificate key. Servers typically select the most secure elliptic curve supported by the client, but may be configured to respect client preference. If the server has a preferred list of curves and the client supports an overlapping set of curves, any connection between the two will use the preferred curve of the server.

The client key exchange message includes the client's public key exchange value on the negotiated curve, which specifies the encoding type and a byte string representing an elliptic curve point.

The premaster secret is computed as the x-coordinate of the ECDH shared secret elliptic curve point. The premaster secret is then used to derive a set of encryption and authentication keys. The client uses the derived keys to authenticate the entire handshake in the client finished message, and the server does the same in the server finished message.

In TLSv1.3, only (EC)DHE key exchange methods are allowed, the keying material is derived from the hash of the entire transcript of the handshake as described in RFC 7627 [68], and the server signs the hash of the transcript with its certificate key, which prevents any type of downgrade attack other than a full man-in-the-middle attack by an attacker who has compromised the server's private certificate key.

3.2.6. ECC in SSH

Elliptic curve use in SSH is specified by RFC 5656 [284]. Elliptic curves can be used in ECDH or ECMQV key exchange and ECDSA for digital signatures. In the SSH handshake, both client and server send a list of supported encryption algorithms in their KEXINIT

				Number of hosts that support...						
Proto	**Port**	**Date**	**BASE**	**ECDHE**	**secp224r1**	**secp256r1**	**secp384r1**	**secp521r1**	**x25519**	**bp256r1**
TLS	443	11/2016	38.6M	24.8M	643.4K (2.6%)	24.1M (97.0%)	5.7M (22.9%)	2.5M (10.2%)	0 (0.0%)	980.1K (3.9%)
	443	08/2017	41.0M	28.8M	811.6K (2.8%)	25.0M (86.9%)	9.1M (31.6%)	2.2M (7.7%)	740.7K (2.6%)	2.4M (8.4%)
SSH	22	11/2016	14.5M	7.9M	0 (0.0%)	7.7M (97.8%)	7.5M (95.6%)	7.5M (95.4%)	6.1M (77.2%)	0 (0.0%)
IKEv1	500	11/2016	1.1M	215.4K	143.8K (66.8%)	211.8K (98.3%)	206.8K (96.0%)	152.8K (71.0%)	0 (0.0%)	0 (0.0%)
IKEv2	500	11/2016	1.2M	101.1K	4.1K (4.1%)	98.2K (97.1%)	98.0K (96.9%)	240 (0.2%)	0 (0.0%)	0 (0.0%)

Table 14: **Server supported curves**—*BASE* gives the number of hosts that we were able to negotiate any key exchange with and *ECDHE* gives the number that support ECDHE key exchange. Percentage support for each curve is with respect to *ECDHE*.

message, and negotiate an algorithm from among the algorithms both support. Supported curves are listed as separate cipher choices for key exchange and signature algorithms. RFC 5656 specifies that SSH implementations must support secp256r1 (nistp256), secp384r1 (nistp384), and secp521r1 (nistp521), and lists 9 additional curves from NIST and SEC2 standards as recommended. Point compression is optional.

If client and server negotiate an ECDH key exchange with a specific curve, the client sends its public key exchange value first. The server then responds with its long-term public host key, its public ECDH key exchange value, and a digital signature using the server's host key over the client and server KEXINIT messages, the server's public host key, the client and server key exchange messages, and the negotiated shared secret. SSH uses ECDH with cofactor multiplication to derive the shared secret.

3.2.7. ECC in IPsec

IPsec uses the Internet Key Exchange (IKE) protocol to negotiate an encrypted and authenticated session. There are two versions of the IKE protocol, IKEv1 and IKEv2. Both rely on Diffie-Hellman key exchange over a set of fixed, standardized groups to negotiate a shared secret. Cremers [102] carried out an automated analysis of the key agreement protocols in IKEv1 and IKEv2 and found a number of vulnerabilities.

The original IKEv1 protocol specified two optional binary curves, ec2n_155 (Oakley Group 3), a 155-bit binary curve, and ec2n_185 (Oakley Group 4), a 185-bit binary curve, among the four groups for Diffie-Hellman key exchange. (The other two were 768-bit and 1024-

bit primes for prime field Diffie-Hellman.) Additional optional binary and prime curves, including the curves from SEC 2, NIST, and Brainpool, have been registered with IANA for IKEv1 and IKEv2 over the course of several RFCs, including RFC 5903 [132], RFC 5114 [206], and RFC 6932 [163].

RFC 2409 specifies that the key exchange value for Oakley groups 3 and 4 consists of the x-coordinate, and the y-coordinate is derived as necessary and not used to derive the shared key. However, RFC 4753 specifies that implementations should send both x and y as the Diffie-Hellman public value and use both in the shared secret.

IKEv1. IKEv1 is specified in RFC 2409. There are two types of handshakes, Main Mode, which requires six messages to establish the connection, and Aggressive mode, which requires three. In main mode, the initiator sends a Security Association (SA) payload, which specifies a collection of cipher suites and Diffie-Hellman groups they support. The responder sends its own SA payload containing its selected cipher suite. The initiator and responder then send key exchange messages for the chosen group. Both parties are then able to compute shared key material, called SKEYID. The computation of SKEYID depends on the authentication method. When signatures are used for authentication, SKEYID = $\text{prf}(N_i|N_r, k_ik_rP)$ where k_ik_rP is the negotiated Diffie-Hellman secret. For the other two authentication methods, public-key encryption and pre-shared key, SKEYID does not depend on the negotiated Diffie-Hellman shared secret, and instead is derived from the cookie or the pre-shared key respectively. Each party authenticates itself by sending an authentication message (AUTH) derived from a hash of SKEYID, the public Diffie-Hellman key exchange messages, the cookies, the initiator's security association, and initiator and responder IDs. In main mode, these authentication messages are encrypted and authenticated using keys derived from the negotiated Diffie-Hellman secret.

In aggressive mode, it is not possible to negotiate the group for Diffie-Hellman. The initiator sends SA and KE messages together, and the responder sends its SA, KE, and AUTH messages together. The initiator finally responds with its AUTH message. The authentication

messages are not encrypted.

IKEv2. IKEv2 combines the SA and KE messages into a single message. The initiator provides a best guess ciphersuite for the KE message. If the responder accepts that proposal and chooses not to renegotiate, the responder replies with a single message containing both SA and KE payloads. Both parties then send and verify AUTH messages, starting with the initiator. The authentication messages are encrypted using session keys derived from the SKEYSEED value which is derived from the negotiated Diffie-Hellman shared secret. The standard authentication modes use public-key signatures over the handshake values.

3.3. Related Work

Bos et al. [74] surveyed elliptic curve adoption rates in 2014, and found that approximately 10% of TLS and SSH hosts supported elliptic curve cipher suites. The ICSI Certificate Notary [175] publishes ongoing statistics on observed SSL/TLS ciphersuites in connections originating from ten research institutes, and reports that at least 88% of connections used ECDHE key exchange in July/August 2017.

Jager, Schwenk, and Somorovsky [179] manually examined ECDH implementations in eight popular TLS libraries in 2015, and found that three of them failed to validate elliptic curve points, leading to full private key recovery for Oracle's default Java JSSE TLS implementation and BouncyCastle. Their analysis was only performed in local test environments. We are unaware of prior work measuring elliptic curve point validation.

Valenta et al. [297] studied prime-field Diffie-Hellman implementations in TLS, SSH, and IPsec in 2016 using both internet-wide scans and source code examination, and found that most examined implementations did not validate subgroup order. Springall, Durumeric, and Halderman [283] measured DHE and ECDHE key exchange reuse among Alexa Top 1 Million domains and found that 1.5% of HTTPS domains supporting ECDHE repeated the same key exchange value in multiple scans, and noted one service that repeated the same key exchange value for 61 days.

Supported Curves	User Agents	Operating Systems	Count
23,24,25	FireFox/46.0-49.0, FitbitMobile/2.28, IE/11.0, uservoice-android-1.2.4, Safari/9.0, Tinder/63105	Win7, Win8, Win10, iOS	1.5M (35.9%)
29,23,24	Chrome/50.0-54.0	Win7, Win8, Win10, Mac OSX, Chrome OS	909.0K (21.7%)
23,24	IE/11, Edge/13.0, Chrome/47.0-51.0	WinVista, Win7, Win8, Win10	661.7K (15.8%)
14,13,25,11,12,24,9,10,22,23,8,6, 7,20,21,4,5,18,19,1,2,3,15,16,17	uservoice-android-1.2.4, Picsart/3.0, okhttp/3.2.0, Playstation/4, Netscape/4.0, Python-urllib/2.7	Win7, Win10, Other	621.3K (14.8%)
25,24,23	SamsungBrowser/2.0-2.1, Wget/1.12	Android, Other	184.4K (4.4%)
23,25,28,27,24,26, 22,14,13,11,12,9,10	Chrome/47.0-53.0, Deluge/1.3.12, Plex Music Agent/1.0, qBittorrent/3.3.7, Transmission/2.84	Win7, Win8, Win10, Other	40.1K (1.0%)
empty	libhttp/3.50, libhttp/4.01, Chrome/25.0	Linux, PlayStation/4	24.9K (0.6%)

Table 15: **Client supported curves extensions with user agents**—We show the ranked list of the most common supported curves lists along with the user agents and operating systems of the clients for a sample of 4,187,201 client hellos collected from Cloudflare. The mapping of curve IDs in the supported curves list to curve names is maintained by IANA [172].

3.4. Elliptic Curve Measurements

In this section, we present our measurements of elliptic curve implementations for TLS, SSH, and IPsec.

3.4.1. Server Curve Support and Preferences

The popularity of different curves varies depending on the protocol. In this section, we describe measurements we performed to understand server curve support for various common ports and protocols, to give a snapshot of elliptic curve deployments.

Server scanning methodology. We performed our scans between November 2016 and August 2017 from the University of Pennsylvania. We used the ZMap [114] Internet-wide scanning tool to perform 10% scans of the IPv4 address space. We extended the Zgrab protocol parser for TLS and SSH to include support for the numerous curves we tested, and used our own Zgrab module for IKEv1 and IKEv2.

For most of our measurements, we scanned a random 10% sample of the public IPv4 Internet on a selection of common ports for TLS, SSH, and IPsec. Unless otherwise specified, the results we present in this paper are extrapolations of our 10% scans to the full IPv4 space, to simplify comparison with other measurements.

For each scan, we first perform a ZMap scan of a randomly selected set of hosts to detect whether a particular port was live. Then, we perform repeated scans of the set of responding hosts using the Zgrab protocol module to detect fine-grained behaviors and support for various cryptographic parameters.

In a TLS and IKE ECDH key exchange, a curve can only be negotiated if it is supported by both the client and the server. To measure support for the elliptic curves shown in Table 14 for TLS, we use Zgrab to perform multiple TLS handshakes, each only offering a single curve at a time in the supported curves extension. For IKE, we offer a security association that includes a curve together with a variety of popular cipher proposal options. In SSH, both the client and server send the list of curves they support, so we can gather curve support from a single scan.

In order to get a baseline measure of support for each protocol, we used scans offering a variety of parameters. The Censys project [2] performs regular 100% TLS and SSH scans using ZMap, so we used their scans from November 2016 and August 2017 as a baseline for support for those protocols. We performed our own 100% IKEv1 and IKEv2 baseline scans.

Server measurement limitations. The survey of Durumeric et al. [115] provides a view of Internet-wide scanning, documenting both the advantages and limitations of the approach. In short, scanning does not allow us to measure hosts that are behind firewalls or are otherwise configured to reject scanning attempts, or hosts whose network operators have requested to be excluded from our scans. Our scans are further restricted to IPv4 hosts, as scanning the IPv6 space efficiently remains an open problem. Despite these limitations, Internet scanning remains an invaluable tool for network operators and defensive security research.

Due to the large number of scans required to measure the selected combinations of server behaviors for our study, we chose to limit each scan to only 10% of the public IPv4 space instead of performing full IPv4 scans. However, we do not expect this to limit the statistical

accuracy of our measurements, although we may occasionally miss rare server behaviors.

Server curve support. ECDH is widely supported by TLS and SSH hosts. We find that 64% of HTTPS hosts and 54% of SSH hosts support ECDH key exchange. As a comparison, Bos et al. [74] report that 7.2% of 30 million HTTPS hosts and 13.8% of 12 million SSH hosts that responded to a ZMap scan in October 2013 supported some form of ECDH key exchange. Adoption of ECDH using common curves for IKE appears to be significantly slower.

Table 14 shows the result of 10% scans extrapolated to full IPv4 scans. We omitted Curve25519 from the November 2016 TLS and IPsec scans since support for this curve was not standardized at the time of the scans. However, we performed additional TLS scans in August 2017 to provide up-to-date numbers on Curve25519 deployment.

The NIST curves secp256r1, secp384r1, and secp521r1 were the most commonly supported curves among servers, but support for each curve varies widely by protocol. secp256r1 was the most popular curve among TLS on port 443, SSH, and IPsec. Support for secp224r1 was surprisingly rare, except for IKEv1. There is a long tail of curve support for other curves in the IANA registries for each protocol; in Section 3.6.1 we give measurements for a number of weak curves.

We performed 10% IKEv1 and IKEv2 scans offering the binary curves ec2n_155 and ec2n_185, but did not detect any hosts that were willing to negotiate these curves. We found two IKE implementations that documented support for these Oakley groups for backwards-compatibility: MikroTik [224] and OpenBSD's iked [244]. We verified that OpenBSD's implementation does indeed support these binary curves by running our scans against an OpenBSD 6.12 instance running in a VM.

TLS also allows servers to specify a custom curve using arbitrary_explicit_prime_curves and arbitrary_explicit_char2_curves. We also performed 10% TLS scans requesting these custom curves, and received no responses on any of the tested ports.

We give full scan data in Appendix 3.B for additional TLS ports.

3.4.2. Client Curve Support and Preferences

Client data methodology. We study client preferences using a sample of client hellos provided by Cloudflare, a popular web performance and security service.

Cloudflare acts as a reverse proxy for web services: when a client connects to a site that uses Cloudflare, a TLS connection is established with a geographically proximal server operated by Cloudflare. This server handles incoming HTTP requests from the client. If a request is for a resource that is cached by Cloudflare, that resource is returned to the client in the response; if the resource is not cached, the Cloudflare server forwards the request to the origin server to obtain a response, which is then returned to the client.

We examined the contents of the TLS client hello together with the client's HTTP user agent string from a uniform sample of incoming HTTPS connections to Cloudflare servers around the world over an approximately 5 minute period on October 17, 2016. 99.4% of the 4.2M client hellos in the sampled traffic included the supported curves extension. At the time of the measurement, Cloudflare was used as an HTTP/HTTPS reverse proxy for over six million domains.

Client measurement limitations. The client dataset that we gathered, while insightful, has multiple limitations. First, the request samples are skewed toward users who were awake and active during the collection period. Collection over a longer period of time might produce a distribution that is more representative of all users. Second, since our data is a raw sample of Cloudflare requests, popular Cloudflare customers are overrepresented in our dataset. Thus, the composition of the data is likely not representative of the web as a whole. We were unable to obtain captured requests from other data sources at the same scale for comparison. Finally, a nontrivial number of requests are from non-browser traffic, including requests from API clients, automated scripts, mobile applications, crawlers, and other bots. This adds depth to the dataset, but means that the dataset does not necessarily reflect the stereotypical view of web traffic as coming exclusively from human-controlled

			Repeats. . .	
Proto	**Port**	**secp256r1**	**Across Hosts**	**By Host**
TLS	443	24.0M	638.7K (2.7%)	5.5M (22.9%)
SSH	22	7.5M	0 (0.0%)	0 (0.0%)
IKEv1	500	168.5K	210 (0.1%)	540 (0.3%)
IKEv2	500	95.1K	800 (0.8%)	1.9K (1.9%)

Table 16: **Repeated key exchanges**—In November 2016, we scanned a randomly selected 10% of IPv4 addresses twice in rapid succession, offering curve secp256r1. *Across Hosts* gives the number of hosts that sent the same key exchange value as another host within a single scan, and *By Host* shows the number of hosts that sent the same key exchange value in both scans.

web browsers.

Client curve support. Table 15 summarizes several of the most common orderings of the supported curves list among sampled clients, using the IANA IDs for each curve. We used Browscap [265] to map software versions to the provided user agent strings. The most common curve preference ordering requests the NIST curves secp256r1, secp384r1, secp521r1 in increasing order of strength, which was provided by a variety of clients. The second most common curve preference ordering in our sample preferred Curve25519, from recent versions of Chrome. The next most common client curve preference ordering in our sample, apparently requested by various APIs, requests most of the curves from SEC 2 in decreasing order of strength.

3.4.3. Repeated Key Exchange Values

For performance reasons, a common behavior among servers is to reuse the same key exchange value for multiple connections, to avoid the need to recompute this value for each client. To detect this behavior, we scan each server twice in rapid succession and check if the key exchange value changes. In Table 16, we offer secp256r1 as the key exchange value, and collect the key exchange values in the server responses.

22% of hosts on TLS port 443 (primarily HTTPS) repeated the same key exchange value in successive scans. 2.6% of TLS port 443 hosts served a non-unique key exchange value that

Client Supported Curve	Server Key Exchange	Hosts
brainpoolp256r1	secp256r1	849.4K
brainpoolp256r1	secp384r1	428
brainpoolp256r1	secp521r1	47
secp224r1	secp256r1	850.0K
secp224r1	secp384r1	474
secp224r1	secp521r1	46
secp256r1	secp384r1	506
secp256r1	secp521r1	49
secp384r1	secp256r1	849.9K
secp384r1	secp521r1	45
secp521r1	secp256r1	849.7K
secp521r1	secp384r1	429

Table 17: **Servers ignoring client supported curves**—In our scans, we found that some servers responded with the same curve regardless of client's list of supported curves. RFC 4492 states that a server must not negotiate the use of an ECC cipher suite if it is not able to complete an ECC handshake with the parameters offered by the client [72].

was shared by at least one other host in the same scan. This could be due to shared hosting providers configured with ephemeral-static key exchange, or random number generation issues.

3.4.4. Other Observations

Our scans uncovered some other interesting server behaviors.

TLS servers ignoring client supported curves. We found that some TLS servers appear to ignore the curves sent in the client supported curves extension, and instead reply with the same curve regardless of whether or not the client indicated support. Across all of the TLS scans we performed in November 2016, we found that 25%, or 8.5M distinct hosts out of 34.6M total hosts returned a server key exchange value specifying a curve that was not present in the client supported curves extension. In Table 17, we show the number of hosts that responded to our scans with an unsupported curve. It appears that these hosts always attempt to negotiate either secp256r1, secp384r1, or secp521r1 rather than terminate the connection when the client offers a curve that they do not support.

In order to understand whether this might be a vulnerability, we experimentally compared responses when our scanner client offered a point on secp256r1 versus the curve that was

originally specified by the client. No servers who sent a point on an incorrect curve accepted a point on the curve that the client originally requested.

Scalar multiplication algorithms. We also performed scans offering points on the twist of the curve. As discussed in Section 3.2.3, TLS implementations do not appear to use single-coordinate ladders for point multiplication, and thus reject points on the twist of the curve. We suspect that hosts that accept invalid curve points but do not accept points on the twist as the client key exchange value are using a mixed-Jacobian scalar-by-point multiplication algorithm, which would cause points on the twist to fail with an arithmetic error but would succeed for points on an invalid curve. However, as shown in Table 20, small numbers of SSH and IKE hosts accepted key exchange values on the twist, suggesting that they may use single-coordinate ladders.

Echo servers. In our IPsec scans, we found that some of the repeated server key exchange values that we observed could be attributed to servers that simply echoed back the same static key exchange value and nonce that we offered in the scan. There were 30 IKEv1 hosts and 25 IKEv2 hosts that exhibited this behavior. These hosts appear to simply echo back an identical copy of any data that they receive. We omit these hosts from the results presented in Table 16.

3.5. CurveSwap Attack

The CurveSwap attack was introduced by Nick Sullivan in 2015 [289]. It is a theoretical attack targeting the curve negotiation to be performed against TLS deployments. Similar in spirit to the FREAK [61] and Logjam [29] attacks, CurveSwap allows a man-in-the-middle to trigger a downgrade attack to force a connection to use the weakest elliptic curve that both parties support. The CurveSwap attack is a parameter negotiation downgrade attack, and can be performed if both client and server support an elliptic curve for which an attacker can break ECDH, either by solving the discrete log or other means. The existence of this attack reduces the overall security of a connection to the security of the weakest elliptic curve supported by both parties.

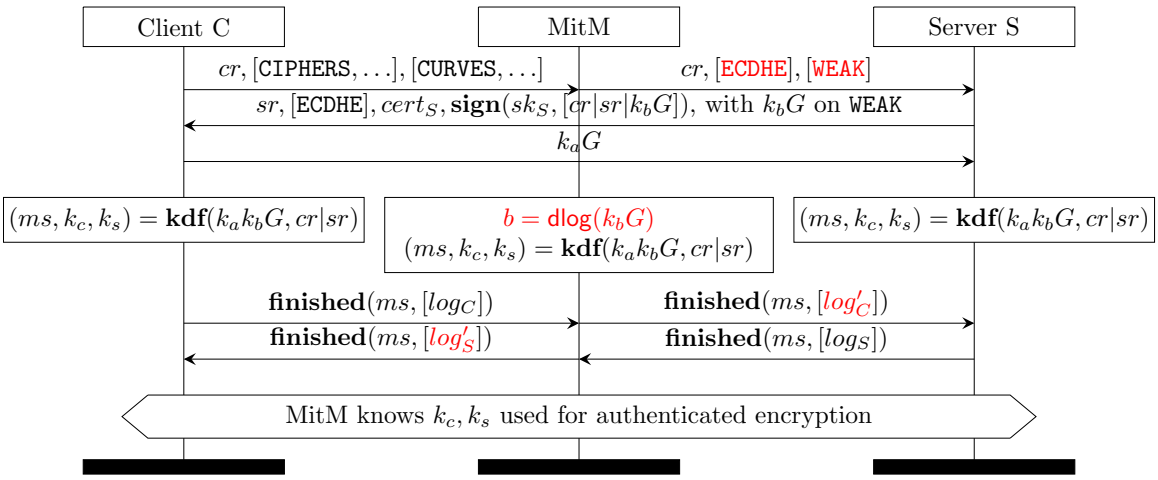


Figure 1: **The CurveSwap attack.**—A man-in-the-middle can force TLS clients to use the weakest curve that both the client and server support. Then, by computing the discrete log on the weak curve, the attacker can learn the session key and arbitrarily read or modify message contents.

3.5.1. CurveSwap for TLS

As explained in Section 3.2.5, a TLS client and server use the supported curves extension [72] to specify which curves each party supports in order to negotiate an elliptic curve group for use in key establishment. The CurveSwap attack demonstrates that in TLSv1.2 and earlier, a man-in-the-middle that can break ECDH for the weakest curve supported by both parties can compromise a connection.

In Figure 1, we depict the CurveSwap attack in a TLS handshake. To mount a CurveSwap attack, the attacker needs to be in a position to man in the middle a connection. When the client sends its client hello message to the server, the attacker replaces it with a client hello message where the client cipher suite list contains only ECDHE cipher suites, and the supported curves extension only contains weak curves.

The server will then reply with its ECDHE public key exchange value on the attacker's chosen weak curve. The attacker passes this message back to the client without modification. The client then replies with its key exchange value on the weak curve. The attacker then computes the elliptic curve discrete log of either the client or server's key exchange message

to compute the client or server's ephemeral private key. At this point, all parties, including the attacker, can then compute the master secret and the session keys. The attacker then intercepts the client and server finished messages and replaces them with finished messages corresponding to the other party's view of the handshake. After the compromised handshake, the client and server have a set of shared session keys that are known to the attacker, allowing the attacker to arbitrarily read and modify messages.

CurveSwap is a vulnerability in the TLS protocol itself, and affects TLS 1.0, 1.1 and 1.2. For these TLS versions, this vulnerability is mitigated somewhat by the TLS Session Hash and Extended Master Secret Extension, described in RFC 7627 [68]. RFC 7267 specifies that the premaster secret is computed from the entire transcript of the handshake, so in the case of an attempted parameter downgrade attack of this form, the attacker would be forced to man in the middle the entire connection instead of merely downgrading it. The CurveSwap attack is mitigated entirely in TLSv1.3, because the server sends a certificate verify message that includes a signature of the entire handshake transcript hash. In order to downgrade the connection, the attacker would need to forge this signature.

3.5.2. CurveSwap for SSH

In SSH, the server uses its long-term host key to sign the entire handshake, including both client and server lists of cipher suites and the negotiated Diffie-Hellman shared secret. Thus a CurveSwap-style attack would require the attacker to compromise the server's host key *and* learn the Diffie-Hellman shared secret. Such a powerful attack does not seem to have any advantage over a complete man-in-the-middle attack.

3.5.3. CurveSwap for IKE

In IKEv1 aggressive mode, it is not possible for the parties to negotiate the Diffie-Hellman group, so a group downgrade attack using aggressive mode is not possible. We note that for the pre-shared key and public-key encryption authentication methods, however, the AUTH messages in aggressive mode do not depend on the negotiated Diffie-Hellman shared secret.

In IKEv1 main mode, both the initiator and responder include the initiator's security as-

sociation (but not the responder's security association) in their AUTH messages, which are encrypted using the negotiated Diffie-Hellman shared secret. An attacker would thus need to learn the Diffie-Hellman shared secret online in addition to compromising the authentication methods used by both parties. There are offline brute-force attacks against pre-shared keys in aggressive mode; documents leaked by Edward Snowden also reference attacks allowing the NSA to learn pre-shared keys in some situations [11, 16, 23].

In IKEv2, authentication is done by having each party sign or MAC their own security association and key exchange messages together with each party's nonces. The initiator and responder's authentication messages are both encrypted and authenticated using the Diffie-Hellman shared secret. Thus a CurveSwap-style downgrade attack would require the attacker to learn the initiator's authentication secret and to learn the Diffie-Hellman shared secret in order to forge the initiator's authentication message online.

3.6. Vulnerability Measurements

We performed a number of large-scale measurements of elliptic curve deployments with a focus on insecure implementation choices that might leave clients or servers vulnerable to CurveSwap.

3.6.1. Brute-forcing Small Curves

The Internet Assigned Numbers Authority (IANA) maintains a registry of valid curves for TLS, which includes several curves at the 80-bit security level [172].

CurveSwap via small curves. The CurveSwap attack allows a man in the middle to downgrade a TLS handshake to use the weakest curve that both the client and the server support. 2^{80} computational work is likely within range for advanced government-level adversaries. However, this amount of computation is quite significant, and is unlikely to be feasible within the timeout of a live TLS handshake in the near future.

However, the widespread use of static-ephemeral key exchange by servers means that a server might reuse its key exchange value for a long enough period to allow an attacker to pre-compute the server's secret exponent for a weak curve. The attacker could then use its

		Repeats. . .	
CurveID	**Support**	**Across Hosts**	**By Host**
ECDHE Hosts	41.0M	–	–
sect163k1	271.7K	2.1K (0.8%)	9.7K (3.6%)
sect163r1	267.8K	230 (0.1%)	7.1K (2.6%)
sect163r2	271.8K	2.1K (0.8%)	10.1K (3.7%)
secp160k1	274.9K	250 (0.1%)	7.7K (2.8%)
secp160r1	276.2K	290 (0.1%)	8.1K (2.9%)
secp160r2	266.9K	360 (0.1%)	7.2K (2.7%)

Table 18: **TLS server support for weak curves**—In August 2017, we scanned a randomly selected 10% of TLS hosts to measure support for weak curves. We scanned each host twice for each curve to detect servers using ephemeral-static keys. The baseline scan shows the number of hosts with which we were able to negotiate any curve. The repeat percentages are with respect to the support scans for each curve.

knowledge of the server's secret exponent for this particular curve to downgrade any clients who support this curve, even if they would normally not prefer it, to this weak curve, and thus be able to decrypt or modify messages during the session.

Weak curve and ephemeral-static measurements. Table 18 shows support statistics for several weak curves, with the number of servers that repeat key exchange values when scanned twice in rapid succession.

We performed additional scans of hosts that initially repeated key exchange values to test the lifespan of ephemeral-static keys. Scanning with curve secp160k1, only 5 hosts responded with the same key exchange value as they did initially after five hours, and only 2 hosts returned the same key exchange value after 25 hours.

We also measure client implementations, and find that a significant number of clients offer weak curves in the supported curves extension. In Table 19, we show that in a sample of over 4 million client hellos collected from Cloudflare, over 16% indicate support for a curve with 80-bit security, opening up these clients to potential CurveSwap attacks. The user agents of these clients indicate that they are mostly API clients rather than browsers.

CurveID	Support
sect163k1	685.6K (16.4%)
sect163r1	682.1K (16.3%)
sect163r2	682.1K (16.3%)
secp160k1	682.6K (16.3%)
secp160r1	682.6K (16.3%)
secp160r2	682.6K (16.3%)

Table 19: **TLS client support for weak curves**—From a sample of 4,187,201 client hellos collected from Cloudflare in October 2016, over 16% offer weak curves in the client hello supported curves extension.

3.6.2. Invalid Curve Attacks

CurveSwap via an invalid curve attack. We now consider the scenario in which a man-in-the-middle attempts to learn the server secret through an invalid curve attack before initiating a CurveSwap attack. In this case, a CurveSwap attack would allow the attacker to force a connection to use a curve for which it already knows the server's ephemeral-static key. Servers are vulnerable to invalid curve attacks when they both fail to validate key exchange parameters and reuse the same ephemeral-static key for multiple connections. If a victim supports a variety of curves, some which are vulnerable to invalid curve attacks, and some which are not, this attack would allow the attacker to downgrade the victim to a vulnerable curve for which they can learn the server's secret.

Measuring invalid curve attacks. We performed extensive measurements to measure the prevalence of implementations vulnerable to invalid curve attacks, and present the results in Table 20. In the end, our scans found evidence of key exchange validation failure and of key reuse, but no hosts that both failed to validate and repeated keys either across hosts or across scans. Thus we do not find evidence of servers vulnerable to invalid curve key recovery attacks.

To test if servers properly validate received client key exchange values, we performed a key exchange using an element of order 5 on an invalid curve for secp256r1. We give the coordinates of this point and the equation of the generator in Appendix 3.A. Table 20 shows

the number of hosts that appeared to accept invalid curve points for the protocols that we scanned.

Since we send an invalid curve point of order 5, the shared secret for the session will be limited to one of five curve elements: (x_1, y_1), $(x_1, -y_1)$, (x_2, y_2), $(x_2, -y_2)$, and infinity. For TLS, SSH, and IKE, only the x-coordinate of the curve element is used as the shared secret for computing the session MAC, so a client sending an invalid point on this curve would have a 2/5 chance of guessing the value correctly by choosing x_1 or x_2 as the shared secret.

In TLS, a client can reach the end of the handshake without authenticating, so in our scans we counted the number of hosts that accepted our client finished message and responded with a server finished message. Thus, we expect the number of hosts that are not properly validating to be 5/2 times as large as the number of hosts that respond with a server finished message. Since Table 20 indicates that 0.31% of HTTPS hosts on port 443 accepted our guessed client finished with our invalid curve point, we estimate that 0.77% of HTTPS hosts fail to perform proper validation.

For SSH and IKE, our scanning methodology does not allow us to reach the end of the handshake without authenticating as a valid client, so we count the number of servers that fail to immediately indicate an error upon receipt of an invalid key exchange value. This does not require us to correctly guess the shared secret, so there is no need to scale the results as for TLS. This also does not account for hosts that perform validation checks later in the handshake, so the numbers presented are an overestimate. In the case of the SSH scans, we show the number of hosts that respond with an ssh_key_exchange_ecdh_reply message after receiving the invalid client public value. All of the SSH hosts that responded to these scans had a protocol banner indicating either "Cerberus", "VShell", or "SshServer". Manually installing CerberusFTPServer_8.0, we were able to replicate this behavior, and found that the server correctly logged an invalid key exchange value in its server logs. This appears to be in violation to RFC 5656, which specifies that the server should validate the client key

Proto	Port	Twist	Invalid	InvalidRepeat
TLS	25	0 (0.0%)	40 (0.0%)	0 (0.0%)
	110	0 (0.0%)	20 (0.0%)	0 (0.0%)
	143	0 (0.0%)	0 (0.0%)	0 (0.0%)
	443	0 (0.0%)	75.5K (0.3%)	0 (0.0%)
	465	0 (0.0%)	260 (0.0%)	0 (0.0%)
	563	0 (0.0%)	10 (0.0%)	0 (0.0%)
	587	0 (0.0%)	0 (0.0%)	0 (0.0%)
	636	0 (0.0%)	150 (0.1%)	0 (0.0%)
	853	0 (0.0%)	20 (1.1%)	0 (0.0%)
	989	0 (0.0%)	0 (0.0%)	0 (0.0%)
	990	0 (0.0%)	230 (0.1%)	0 (0.0%)
	992	0 (0.0%)	10 (0.0%)	0 (0.0%)
	993	0 (0.0%)	8.1K (0.3%)	0 (0.0%)
	994	0 (0.0%)	10 (0.4%)	0 (0.0%)
	995	0 (0.0%)	6.7K (0.2%)	0 (0.0%)
	8443	0 (0.0%)	19.2K (1.5%)	0 (0.0%)
SSH	22	4.1K (0.1%)	3.3K (0.0%)	0 (0.0%)
IKEv1	500	530 (0.2%)	500 (0.2%)	0 (0.0%)
IKEv2	500	4.1K (4.0%)	4.1K (4.0%)	0 (0.0%)

Table 20: **Invalid key exchanges**—In November 2016, we scanned a randomly selected 10% of IPv4 addresses offering order 5 points on an invalid curve and on the twist of curve secp256r1. We show the number of hosts for which handshake negotiation is successful. As described in Section 3.6.2, we estimate that the number of vulnerable TLS hosts is 5/2 times larger than the numbers reported in the table. For SSH and IKE, these numbers are an upper bound on the number of vulnerable hosts.

exchange before sending its own key exchange value.

3.6.3. Twist Attacks

CurveSwap via twist attacks. We now investigate an attack vector that exploits the fact that there are several standardized curves with weak twist security. For example, an invalid curve attack using the twist for secp224r1 can be used to recover the secret key in only $2^{58.4}$ work, compared to its expected 112-bit security level [58].

Consider a server that uses a single-coordinate ladder for scalar-by-point multiplication, such as the Montgomery or Brier-Joye ladders. Single-coordinate ladders operate on only the x-coordinate of the key exchange value, making it impossible to specify a point on an invalid curve [78, 227]. However, an attacker can send an x-coordinate that does not

correspond to a point on the negotiated curve, but does lie on the twist of the curve. If the server employs a single-coordinate ladder for scalar-by-point multiplication, then the server will compute the shared secret as a point on the twist of the curve. For curves with a weak twist, the attacker can send low-order points on the twist, and carry out a small subgroup attack to reconstruct the server's ephemeral-static key. To prevent this attack, an additional check is required to ensure that the specified x-coordinate lies on the curve, and not the twist of the curve.

There are a number of curves with weak twists that bring twist attacks into feasible range [58, 128]. Notably, in addition to the NIST-standardized secp224r1, brainpoolp256t1 also has a weak twist, with an attack cost of 2^{44}. secp256r1 is secure against twist attacks with an attack cost of 2^{120}.

Measuring twist attacks. To test for this behavior, we perform scans sending a point in the subgroup of order 5 on the twist of secp256r1 as the client key exchange value. We chose secp256r1 because it has the highest support among the protocols we studied. We give the point coordinates and the twist equation in Appendix 3.A. The scan results, presented in Table 20, indicate that no hosts accepted points on the twist of the curve. To test if point compression influenced server behavior, we performed an additional 10% scan of TLS on port 443 sending a compressed point of order 5 on the twist of secp256r1, and found that no hosts accepted this key exchange value.

We suspect that hosts accepting invalid curve points but not accepting points on the twist as the client key exchange value are using a mixed-Jacobian scalar-by-point multiplication algorithm, which would cause points on the twist to fail with an arithmetic error but would succeed for points on an invalid curve.

3.7. Source Code Analysis

We examined a number of libraries to understand their elliptic curve implementations, and found multiple vulnerabilities. We also described our findings in a blog post [274].

Library	Language	ECDH Support	Status
cjose	C/C++	No	–
jose-jwt	Haskell	No	–
jose4j	Java	Yes	fixed v0.5.5
Nimbus JOSE+JWT	Java	Yes	fixed v4.34.2
Apache CXF	Java	Yes	not vuln.
json-jwt	Ruby	No	–
phpOIDC	PHP	No	–
jose-php	PHP	No	–
js-jose	Javascript	No	–
go-jose	Go	Yes	fixed v1.0.4
jose2go	Go	Yes	fixed v1.3
node-jose	node.js	Yes	fixed v0.9.3

Table 21: **JWE libraries**—We manually inspected the source code of several libraries implementing JSON Web Encryption, and found that many were vulnerable to a classic invalid curve attack.

3.7.1. Failure to Validate in JSON Web Encryption Standards and Implementation

We examined the source code of many libraries implementing RFC 7516, JSON Web Encryption (JWE), focusing on the Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES) algorithms. The complete list of libraries that we examined is available in Table 21. We found that many of these libraries were vulnerable to a classic invalid curve attack as described in Section 3.2.4. This would allow an attacker in the role of a sender to completely recover the secret key of the receiver. Almost all the implementations we examined failed to validate that the received public key, contained in the `JWE Protected Header`, is on the curve. Although they did not validate the recieved public key before performing the scalar multiplication, some of the libraries that we examined (Nimbus JOSE+JWT, jose4j) were protected from the invalid curve attack by Java's BouncyCastle or up-to-date Java Sun JCA elliptic curve library, which includes a check that the result of the scalar multiplication is on the curve. However, libraries implemented in languages without this additional check, such as Cisco's node-jose and jose2go, were completely vulnerable. As shown in Table 21, we reported the vulnerabilities to library maintainers to ensure that implementations included the check that incoming public keys are on the agreed-upon curve. The go-jose vulnerability was found and reported by Nguyen [236].

3.7.2. Bug in NSS/Java in Elliptic Curve Addition

Both NSS and Java use the 5-bit window NAF method for scalar-by-point multiplication from [81]. Both implementations missed a critical `if/else` statement that lead the calculations to produce incorrect results on some inputs. In particular, there exist values of the scalar for which the algorithm would yield the point at infinity as a result while the actual correct result should be a finite value. We were unable to figure out a way to exploit this flaw.

We disclosed these flaws to Mozilla and Oracle in March 2017. The flaw was patched by including the missing `if/else` statement [232, 248].

3.8. Discussion

Although we found some vulnerable, buggy, and non-compliant elliptic curve behavior in most of the protocols we measured, the fact that these behaviors do not appear to lead to a full CurveSwap attack is good news. (The exception is JWE, where the invalid curve attacks are devastating and do not require a parameter downgrade.)

3.8.1. Complexity of Curve Support

We observe that there are a large number of curves that are supported in the protocols we studied, some of them dating from much earlier in the study of elliptic curves before different varieties of implementation attacks were as well understood. While having many curve sizes or parameter types would seem to give protocols and implementations room to adapt their speed and security needs, support for many of these curves risks becoming a liability if attacks on some classes improve enough to allow a feasible CurveSwap attack in TLS or other protocols. In addition, enumerating the current state of different attacks on each curve is quite complex [58].

While recent curve constructions such as Curve25519 are designed to be as resistant to implementation mistakes as possible, the move to "new" algorithms such as single-coordinate ladders, which appear from our data not to be widely implemented for most curves, will likely result in the discovery of new bugs of the type we discovered in NSS/Java.

3.8.2. Protocol Security

The recent spate of cipher downgrade, transcript mismatch, and message forwarding attacks against TLS has highlighted the need for protocol-level protections against these types of man-in-the-middle attacks. Fortunately, TLSv1.3 includes multiple layers of handshake downgrade protection, including client and server authentication of the entire transcript hash using long-term secrets when possible, and computing session keys from the entire transcript. We note that the SSH protocol builds in such protection by having the server sign the entire transcript, as does IKE when using signature authentication. We hope that the community's improved understanding of protocol security means that downgrade attacks are a thing of the past.

3.A. Invalid Curve and Twist Points

We tested for curve validation in secp256r1 by using a generator of a subgroup of order 5 on the curve $y^2 = x^3 + ax + (b - 1)$ with a and b as specified in [80] for secp256r1. The coordinates of our generator were

$$x = \texttt{BFD3 5739 ED4B 4D93 8C91 E835 7C7E C4C4 1DE9 FDFC}$$
$$\texttt{1669 88EB D1DF A09C 7959 6661}$$
$$y = \texttt{8949 2141 E9E8 1674 9798 62D9 FC62 21C4 A672 B890}$$
$$\texttt{33E0 7B86 DA40 D67D 5C0F 53E3}$$

We tested for twist validation in secp256r1 by sending a point of order 5 on the twist $y^2 = x^3 + a'x + b'$ with

$$a' = \texttt{8EB0 E29E C8A5 CCCB 65B9 936F B5B2 67E6 57D4 83DB}$$
$$\texttt{CDC0 2A88 8A7F 72E8 935B B316}$$
$$b' = \texttt{2F9B 5262 887E 1766 8BBA F58E 54B8 2E42 C72E D167}$$
$$\texttt{21BD 3325 DEB7 9B62 ADE7 4BD6}$$

The coordinates of our generator were

$$x = \texttt{8FB5 0654 3387 E96C D244 8468 9BF6 CC0C F383 4F33}$$
$$\texttt{D8CD 6442 4B11 7D3B ECA1 E0B5}$$
$$y = \texttt{E042 260E 3A00 30A5 5B46 8D2A DEBA D3D4 B613 373C}$$
$$\texttt{0C38 FCD8 5434 C2B8 B7F7 C1EA}$$

3.B. Extended Scans on Multiple Ports

We extended our scans to a variety of ports where TLS is used to secure services such as IMAP, POP3, SMTP, LDAP, and more.

			Repeats...	
Proto	**Port**	**secp256r1**	**Across Hosts**	**By Host**
TLS	25	375.8K	590 (0.2%)	21.9K (5.8%)
	110	126.3K	190 (0.2%)	290 (0.2%)
	143	120	0 (0.0%)	0 (0.0%)
	443	24.0M	638.7K (2.7%)	5.5M (22.9%)
	465	2.6M	156.1K (6.1%)	60.3K (2.3%)
	563	45.5K	36.4K (79.8%)	37.7K (82.7%)
	587	310.4K	160 (0.1%)	160 (0.1%)
	636	119.5K	39.1K (32.7%)	77.7K (65.0%)
	853	1.7K	40 (2.4%)	840 (50.6%)
	989	1.8K	80 (4.4%)	1.1K (61.3%)
	990	245.6K	24.1K (9.8%)	39.3K (16.0%)
	992	28.4K	40 (0.1%)	980 (3.5%)
	993	771.3K	55.0K (7.1%)	83.2K (10.8%)
	994	2.2K	100 (4.5%)	1.0K (45.9%)
	995	717.1K	57.4K (8.0%)	79.6K (11.1%)
	8443	1.3M	49.3K (3.9%)	274.6K (21.7%)
SSH	22	7.5M	0 (0.0%)	0 (0.0%)
IKEv1	500	168.5K	210 (0.1%)	540 (0.3%)
IKEv2	500	95.1K	800 (0.8%)	1.9K (1.9%)

Table 22: **Repeated key exchanges**—Extended version of Table 16.

Proto	Port	Twist	Invalid	InvalidRepeat
TLS	25	0 (0.0%)	40 (0.0%)	0 (0.0%)
	110	0 (0.0%)	20 (0.0%)	0 (0.0%)
	143	0 (0.0%)	0 (0.0%)	0 (0.0%)
	443	0 (0.0%)	75.5K (0.3%)	0 (0.0%)
	465	0 (0.0%)	260 (0.0%)	0 (0.0%)
	563	0 (0.0%)	10 (0.0%)	0 (0.0%)
	587	0 (0.0%)	0 (0.0%)	0 (0.0%)
	636	0 (0.0%)	150 (0.1%)	0 (0.0%)
	853	0 (0.0%)	20 (1.1%)	0 (0.0%)
	989	0 (0.0%)	0 (0.0%)	0 (0.0%)
	990	0 (0.0%)	230 (0.1%)	0 (0.0%)
	992	0 (0.0%)	10 (0.0%)	0 (0.0%)
	993	0 (0.0%)	8.1K (0.3%)	0 (0.0%)
	994	0 (0.0%)	10 (0.4%)	0 (0.0%)
	995	0 (0.0%)	6.7K (0.2%)	0 (0.0%)
	8443	0 (0.0%)	19.2K (1.5%)	0 (0.0%)
SSH	22	4.1K (0.1%)	3.3K (0.0%)	0 (0.0%)
IKEv1	500	530 (0.2%)	500 (0.2%)	0 (0.0%)
IKEv2	500	4.1K (4.0%)	4.1K (4.0%)	0 (0.0%)

Table 23: **Invalid key exchanges**—Extended version of Table 20.

				Number of hosts that support...						
Proto	**Port**	**Date**	**BASE**	**ECDHE**	**secp224r1**	**secp256r1**	**secp384r1**	**secp521r1**	**x25519**	**bp256r1**
TLS	25	11/2016	–	1.0M	420 (0.0%)	1.0M (99.7%)	3.1K (0.3%)	220 (0.0%)	0 (0.0%)	0 (0.0%)
	110	11/2016	–	182.7K	270 (0.1%)	176.7K (96.7%)	125.3K (68.6%)	113.6K (62.2%)	0 (0.0%)	580 (0.3%)
	143	11/2016	–	130	0 (0.0%)	130 (100.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
	443	11/2016	38.6M	24.8M	643.4K (2.6%)	24.1M (97.0%)	5.7M (22.9%)	2.5M (10.2%)	0 (0.0%)	980.1K (3.9%)
	443	08/2017	41.0M	28.8M	811.6K (2.8%)	25.0M (86.9%)	9.1M (31.6%)	2.2M (7.7%)	740.7K (2.6%)	2.4M (8.4%)
	465	11/2016	–	2.7M	21.6K (0.8%)	2.7M (99.9%)	230.4K (8.4%)	213.2K (7.8%)	0 (0.0%)	2.0K (0.1%)
	563	11/2016	–	45.7K	60 (0.1%)	45.7K (99.9%)	2.9K (6.3%)	1.6K (3.6%)	0 (0.0%)	280 (0.6%)
	587	11/2016	–	836.9K	20 (0.0%)	836.6K (100.0%)	330 (0.0%)	40 (0.0%)	0 (0.0%)	0 (0.0%)
	636	11/2016	–	121.0K	2.8K (2.3%)	120.8K (99.8%)	43.5K (36.0%)	10.7K (8.8%)	0 (0.0%)	1.1K (0.9%)
	853	11/2016	–	1.8K	60 (3.4%)	1.7K (97.2%)	1.2K (66.5%)	400 (22.7%)	0 (0.0%)	240 (13.6%)
	989	11/2016	–	1.9K	30 (1.6%)	1.8K (98.9%)	1.3K (69.9%)	280 (15.1%)	0 (0.0%)	140 (7.5%)
	990	11/2016	–	246.4K	1.3K (0.5%)	243.7K (98.9%)	202.1K (82.0%)	184.1K (74.7%)	0 (0.0%)	690 (0.3%)
	992	11/2016	–	28.5K	300 (1.1%)	28.5K (99.8%)	27.7K (97.1%)	26.8K (93.9%)	0 (0.0%)	300 (1.1%)
	993	11/2016	–	2.9M	31.8K (1.1%)	772.8K (26.5%)	2.6M (89.0%)	380.2K (13.0%)	0 (0.0%)	97.9K (3.4%)
	994	11/2016	–	2.5K	100 (4.0%)	2.3K (94.3%)	1.6K (63.2%)	510 (20.6%)	0 (0.0%)	260 (10.5%)
	995	11/2016	–	2.8M	24.5K (0.9%)	717.9K (25.9%)	2.5M (89.0%)	359.5K (13.0%)	0 (0.0%)	88.6K (3.2%)
	8443	11/2016	–	1.3M	102.4K (7.9%)	1.3M (98.9%)	406.9K (31.5%)	159.5K (12.4%)	0 (0.0%)	22.1K (1.7%)
SSH	22	11/2016	14.5M	7.9M	0 (0.0%)	7.7M (97.8%)	7.5M (95.6%)	7.5M (95.4%)	6.1M (77.2%)	0 (0.0%)
IKEv1	500	11/2016	1.1M	215.4K	143.8K (66.8%)	211.8K (98.3%)	206.8K (96.0%)	152.8K (71.0%)	0 (0.0%)	0 (0.0%)
IKEv2	500	11/2016	1.2M	101.1K	4.1K (4.1%)	98.2K (97.1%)	98.0K (96.9%)	240 (0.2%)	0 (0.0%)	0 (0.0%)

Table 24: **Server supported curves**—Extended version of Table 14.

CHAPTER 4 : Side-channel attack against Curve25519

4.1. Introduction

Since their introduction over a decade ago [54, 250, 254], microarchitectural attacks [136] have become a serious threat to cryptographic implementations. A particular threat arises from asynchronous attacks, where the attacker only has to execute a program concurrently with the victim's program (on the same physical CPU) in order to collect temporal information about the victim's behavior. With this temporal information at hand, the attacker can recover the internal workings of the victim.

Because microarchitectural attacks execute on the same processor as the victim, the attacker can only achieve limited temporal resolution. Typically, the attacker can only distinguish between event timings if the events are several hundreds or thousands of execution cycles apart. Consequently, past asynchronous attacks often target key-dependent variations in either the order of high-level operations or in their arguments. More specifically, such attacks usually target the square-and-multiply sequence of the modular exponentiation in RSA [254, 308], ElGamal [209, 316] and DSA [256], or the equivalent double and add sequence of scalar-by-point multiplication in ECDSA [34, 52, 263, 307]. A notable exception is the attack of Pereida García and Brumley [255], which targets the modular inversion used in ECDSA.

With the increased sophistication of microarchitectural attacks, many implementations of cryptographic algorithms have had their side channel robustness investigated, analyzed, and improved. Dealing away with obvious side channel vulnerabilities such as multiplication operations on every set key bit and key-dependent table access, existing implementations have been replaced with more regular algorithms, while newer schemes are designed with side channel resistance in mind from the start.

For elliptic curve cryptography, one approach for reducing the leakage from the scalar-by-point multiplication is to use the Montgomery powering ladder [227]. Performing one point

addition operation and one point doubling operation per key bit, regardless of the value of the bit, makes the Montgomery ladder much more resilient to side channel attacks compared to other scalar-by-point multiplication algorithms [184, 243]. Side channel resistance can be further improved by using unified addition formulas, which eliminate operand-dependent branches [71] from point addition and point multiplication routines.

Since modern cryptographic algorithms and implementations have almost completely eliminated high-level key-dependent branches and memory accesses, our work studies the side channel implications of low-level branches typically performed deep inside basic integer arithmetic operations, such as modular reductions.

4.1.1. Our Contribution

In this paper, we present a new microarchitectural key extraction attack on a highly-regular real-world implementation of Curve25519 [55]. We show that the specific mathematical structure and recommendations of use for many recently suggested elliptic curves (including Curve25519) actually allow for an *easier* exploitation of low-level side channel weaknesses.

We empirically demonstrate our attack using three real-world applications of Curve25519: git-crypt [44], a git plugin for encrypting git repositories; Pidgin-OpenPGP [149], a plugin for the Pidgin chat client for encrypting chat messages; and Enigmail [266], a popular Thunderbird plugin for email encryption. All of these applications use Libgcrypt [152] as their underlying cryptographic library. Since Libgcrypt's implementation of Curve25519 uses the Montgomery ladder for scalar-by-point multiplication, branchless formulas for point doubling and addition, and built-in countermeasures specially designed to resist cache attacks, our attack cannot observe high-level key dependent behavior, such as key-dependent branches or memory accesses. Instead, we achieve key extraction by combining the specific mathematical structure of Curve25519 with low-level side channel vulnerabilities deep inside Libgcrypt's basic finite field arithmetic operations. By observing the cache access patterns during at most 11 scalar-by-point multiplications, our attack recovers the entire secret scalar within a few seconds. We note that the mathematical structure that enables our

attacks in also present in other popular curves such as Curve41417 [60] and Curve448 [161] (Goldilocks curve) when represented in Montgomery form [227].

The Dangers of Order-4 Elements. To extract the secret key, our attack uses side channel leakage produced during decryption with low-order elements, which are present in many recently designed curves. While the side channel risks of order-2 elements are known [139, 309], attacking Montgomery ladder implementations using an order-2 element fails to produce key extraction (see Section 4.1.2). Instead, our attack takes advantage of the side channel leakage produced by decrypting with an order-4 element. The risks of such elements have been suggested in the past [125], however we are not aware of any demonstration of a practical attack on elliptic curve cryptography that exploits elements of order 4.

The Shortcomings of Existing Countermeasures. Many recently designed elliptic curves support scalar-by-point multiplication using single-coordinate ladders, which forces all received inputs x to be either on the curve or on the "twist" of the curve. Moreover, these curves are also twist-secure, meaning that the twist is also resistant to small subgroup attacks. While these properties mitigate many invalid-curve attacks [55], they also lead implementations to omit *all* input validation, causing them to perform secret-key operations on potentially adversarial inputs. Indeed, while the recommendation to avoid performing validation [56] makes sense in the original context of a carefully designed, constant-time implementation that does not contain side channel weaknesses and is not vulnerable to small subgroup attacks, we argue that this validation improves side-channel resistance for implementations that might not be as carefully designed and implemented for constant-time side-channel resistant execution. This is because the absence of input validation leaves the door open for exploiting other potential side-channel vulnerabilities, as we show in this paper.

Even when countermeasures against low order elements and small subgroup attacks exist, they often do *not* prevent all side-channel attacks. For example, RFC 7748 [200] recom-

mends "ORing all the bytes (of the output) together and checking whether the result is zero, as this eliminates standard side-channels in software implementations." One reason that this countermeasure does not work against side-channel attacks that exploit low-order elements is that it is enacted *after* the scalar-by-point multiplication has been performed, when the leakage is already obtained by the adversary.

Thus, we suggest that implementations reject low-order elements *before* performing sensitive secret key operations, in addition to deploying other side channel countermeasures such as point blinding and exponent randomization. See Section 4.6 for details.

4.1.2. Attack Description

We target the ECDH public-key encryption algorithm, as specified in RFC 6638 [182] and NIST SP800-56A [49] and implemented in OpenPGP [86]. We demonstrate our attack on applications that use Libgcrypt, the underlying cryptographic library of GnuPG [152]. The ECDH decryption operation primarily consists of multiplying the secret key (a scalar) by a curve point. For the case of ECDH encryption using Curve25519, Libgcrypt performs the scalar-by-point multiplication using a Montgomery ladder implementation with a single branchless formula for simultaneously computing point addition and doubling. As a protection from cache attacks, Libgcrypt also contains carefully designed routines for performing the swap operations needed to implement the Montgomery ladder. Thus, for every bit of the secret scalar, Libgcrypt performs the same fixed sequence of operations that do not contain any high-level operand-dependent branches or memory accesses.

Unlike traditional Weierstrass and Koblitz curves (such as P192, P224, P256, P384, P521 and Secp256k1) many newly designed curves (such as Curve25519, Curve41417 and Curve448) can be represented in Montgomery form [227] to obtain additional performance speedups. We observe that for a curve to be representable in Montgomery form, it must have an order that is a multiple of four, implying that it contains low-order elements such as an order-2 element $\mathtt{G}_2$ and in many cases an order-4 element $\mathtt{G}_4$. While the existence of order-2 elements is a known side channel risk [125, 139, 309], this risk is slightly mitigated for Montgomery

curves using a Montgomery ladder based implementation since the order-2 element is the point of origin ($x = 0, y = 0$). When this point is passed into many implementations of the Montgomery ladder, it causes the result and all computed intermediate values to be zero, irrespective of the secret key [125, 267].

Instead, we perform the ECDH decryption operation using an order-4 element $\mathtt{G}_4$, and take advantage of its representation in projective coordinates. As our analysis in Section 4.3 shows, using a Montgomery ladder for decrypting $\mathtt{G}_4$ results in curve points of particular mathematical structure appearing as intermediate values during the decryption process. Thus, while the operations performed by the Montgomery ladder scalar-by-point multiplication routine are fixed, our attack links the *operands* of these operations to the secret scalar. Exploiting a side channel weakness in Libgcrypt's modular reduction operation via a cache side channel, we can observe this link and recover the secret scalar.

4.1.3. Targeted Software and Current Status

In this paper, we focus on the ECDH decryption operation and the Montgomery ladder scalar-by-point multiplication routine as implemented in Libgcrypt. We used Libgcrypt version 1.7.6 (which is the latest version of Libgcrypt at the time of writing) as supplied as part of the latest Ubuntu 17.04.

We have disclosed our findings to the GnuPG team and are working with them to implement countermeasures against our attack. The vulnerability has been assigned CVE-2017-0379.

4.1.4. Attack Scenarios

Libgcrypt is part of the GnuPG code base [152], and is used in particular by GnuPG 2.x, a popular implementation of the OpenPGP standard [86] for encrypting files and emails. While our attack requires the decryption of a specific adversarial input (an order-4 element), Libgcrypt is used as the cryptographic back-end for many applications, and as such is often supplied with externally controlled inputs. See [151] for a list of supported Libgcrypt front ends. In Section 4.4.3, we detail our attacks against the following three front-end applications:

Enigmail. For an attack on encrypted email we use the Thunderbird plugin Enigmail. As Genkin et al. [140] observe, Enigmail automatically decrypts incoming emails by passing them to GnuPG, which uses Libgcrypt as its cryptographic engine. To attack Enigmail, we inject an element of order 4 into Libgcrypt we send the victim a PGP/MIME-encoded e-mail [120], with the element of order-4 as the ciphertext.

Git-crypt. Git-crypt is a git plugin for encrypting files uploaded to git repositories. The aim is to allow for uploading content to a public repository and only authorize a select group of users to access the uploaded content. The user specifies the files to be encrypted, with encryption taking place automatically when pushing changes to the repository. The files are automatically decrypted when changes are pulled from the repository. Git-crypt uses a hybrid encryption scheme. Repository files are encrypted with a randomly-generated AES key. For each authorized user, git-crypt encrypts the AES key with the user's public key and stores the encrypted AES key in the repository. An attacker can thus create a malicious key file with an order-4 element as the ECDH public value in the ciphertext. Uploading this file as the victim's encrypted key file. When the victim pulls the repository, git-crypt automatically tries to decrypt the repository, resulting in an order-4 element being injected into Libgcrypt's scalar-by-point multiplication routine.

Pidgin-OpenPGP. Pidgin is a popular open-source chat application that supports communication across a variety of chat networks [18]. The Pidgin-OpenPGP plugin allows users to encrypt and sign their chat messages using GnuPG [149]. For the attack on Pidgin-OpenPGP, we generate an encrypted chat message and replace the ciphertext with the element of order 4. When the victim receives the message, Pidgin-OpenPGP automatically tries to decrypt it, triggering the attack.

4.1.5. Attack Feasibility and Limitations

The specific attack that we describe in this paper is realistic in settings where the attacker can share memory with the victim. In particular, we have tested the attack when the attacker process is running as a separate user within the same operating system as the victim

process. The Flush+Reload technique we use has also been shown to be effective in PaaS cloud environments, where the attacker and the victim execute within two different containers [317] and in virtualized environments that use memory de-duplication [176, 308]. In these settings, monitoring cache activity during the decryption of only a few chat messages, emails, or git pulls will be sufficient to extract the victim's secret key.

When the attacker and the victim do not share memory, our specific attack does not work. However, we note that use the LLC Prime+Probe attack [209] does not require memory sharing and has been shown effective in cloud environments [174]. Hence, avoiding memory sharing does not guarantee protection.

4.1.6. Related Work

In this section, we review classes of side-channel attacks that built the foundations for our work.

Physical Side Channel Attacks on ECC Running on Small Devices. Since the first (simulated) attacks of Coron [99], there have been numerous physical side channel key extraction attacks on implementations of elliptic curve cryptography running on small devices. See the surveys [123, 124] and the references therein. However, most of these results either attack naive implementations which contain key-dependent branches (such as the double-and-add algorithm) or take advantage of subtle effects which are only visible at bandwidths exceeding the device's clock rate and are thus impossible to observe using low-bandwidth channels such as the cache side channel.

Two exceptions to the above approach are the Refined Power Analysis attack of Goubin [155] and the Zero-Value Point Attacks of Akishita and Takagi [31] which do seem to use low-bandwidth-observable effects. However both of these attacks require obtaining measurements during the decryption of hundreds of adaptively chosen ciphertexts in order to perform key extraction, making them easily detectable.

Physical Side Channel Attacks on ECC Running on Complex Devices. Key

extraction attacks against elliptic curve cryptography implementations running on complex devices have also been demonstrated using both cache and physical side channels. More specifically, electromagnetic key extraction attacks were demonstrated by Genkin et al. [143] on GnuPG's ECDH encryption using a double-and-add 1NAF implementation running on PCs and by Genkin et al. [144] and Belgarric et al. [51] for ECDSA signing routine executed on smartphones.

Attacks on Curve25519. Kaufmann et al. [190] describe an attack on an implementation of Curve25519, which shows timing variations when compiled with the Microsoft Visual C compiler. The attack requires 25000 chosen ciphertexts per each key bit and takes about a month to recover the key.

Duong [111] describes a theoretical attack against Diffie-Hellman with Curve25519 which exploits the lack of public key validation. The attack assumes an adversary that can replace public keys with the element at infinity, in which case the shared secret will be known.

Software-based Side Channel Attacks on Cryptography Running on PCs. Attacks on PC implementations of cryptography have also been demonstrated using software channels such as the timing channel [83, 84]. Starting with [54, 250, 254, 292, 293] cache attacks have been extensively used to break implementations of cryptographic primitives running on PCs. See Ge et al. [136] for a survey. Brumley and Hakala [82] perform a cache attack on an implementation of ECDSA. The Flush+Reload technique we use has been used for attacks on RSA [308], AES [158, 176], ECDSA [34, 52, 263, 307] and BLISS [156]. The attacks of [28, 307] are of special relevance as they are the only prior works to use microarchitectural attacks to break an implementation that uses the Montgomery ladder. Their attacks, however, exploited a high-level conditional statement that does not exist in the Libgcrypt implementation of the ladder.

Side Channel Attacks on GnuPG. Starting with [140, 235, 308], GnuPG has been targeted by various key extraction attacks. These include attacks on GnuPG's RSA and

ElGamal implementations [139, 140, 141, 142, 209, 308] as well as attacks on GnuPG's ECDH encryption [143] and ECDSA signatures implementations [52, 263]. We note that the attacks of [52, 143, 263] are not applicable to the implementation of Montgomery ladder based ECDH encryption that we attack in this paper; after version Libgcrypt 1.6.5, GnuPG no longer uses the Double-and-Add 1NAF implementation attacked by [143], and the attacks of [52, 263] that mount a lattice attack on ECDSA using partially known nonces are not applicable for ECDH.

Attacks Using Low-Order Elements. The risk of performing public key cryptographic operations on elements of low order has been previously demonstrated on various types of public key encryption methods. Yen et al. [309] and Genkin et al. [139] achieve key extraction by using an order-2 element as a chosen ciphertext with implementations of RSA and ElGamal that are based on the square-and-always-multiply exponentiation algorithm. For Elliptic Curve Cryptography, low-order elements have been used for mounting invalid point attacks [70, 207] as well as for fault injection attacks [125]. More specifically, Fan et al. [125] present a theoretical fault injection attack against elliptic-curve Diffie-Hellman key exchange operating over NIST curves, which do not have low-order elements. The attack starts by performing a Diffie-Hellman key exchange using a valid curve point with a short Hamming distance to a point of low order on a twist of the curve. Next, the attacker can (theoretically) inject a carefully-timed fault in the hope of flipping bits in the point's coordinates thus causing the implementation to perform a scalar-by-point multiplication operation with a low-order element on the twist. While Fan et al. [125] do not empirically demonstrate their attack, they do argue, similar to our analysis in Section 4.3, that the leakage (via physical side channels) resulting from performing the scalar-by-point multiplication with a low order point (order-4 or order-2) should contain enough information to reveal the secret key.

4.2. Preliminaries

4.2.1. Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) is an approach to public-key cryptography using elliptic curves over finite fields. The underlying hardness assumption in ECC schemes is the Elliptic Curve Discrete Logarithm Problem (ECDLP): given an elliptic curve group $\mathbb{G}$, a generator $\mathtt{G}$, and a point $\mathtt{P}$ it is assumed to be hard to find a scalar k satisfying $\mathtt{P} = [k]\mathtt{G}$. (Here and onward, we use additive group notation, and $[k]\mathtt{G}$ denotes scalar-by-point multiplication further described in Section 4.2.2 below.) The running time of the best known algorithm for solving ECDLP (without the presence of side channel leakage) is linear with the square root of the order of the subgroup generated by the elliptic curve's generator.

Curve Formulas. Elliptic curves can be expressed with several different representations. The traditional model for elliptic curves is the Weierstrass equation $y^2 = x^3 + ax + b$. Every elliptic curve over a finite field $\mathbb{F}_p$ of a prime order can be converted to this form. Some widely-used examples of curves expressed in this form are the NIST curves from FIPS 186-4 [239] and the Brainpool curves [219].

Alternative elliptic curve representations are often used for speed. Montgomery [227] introduced the eponymous Montgomery form elliptic curves, which are specified using the curve shape $By^2 = x^3 + Ax^2 + x$. A main advantage of curves of this form is that scalar-by-point multiplication can be implemented using only the x coordinate. The single-coordinate version of the Montgomery ladder algorithm for scalar-by-point multiplication requires fewer arithmetic operations than standard Weierstrass scalar-by-point multiplication methods while offering better side channel resistance [184, 243]. The most widely used curve of this form is Curve25519, which was introduced by Bernstein [55]. Other curves that can be specified in this form include Curve41417 [60] and Curve448 [161] (the Goldilocks curve).

Domain Parameters and Cofactors. An elliptic curve group is defined by a set of domain parameters which consists of the following values: p, a prime which defines the

prime-order finite field $\mathbb{F}_p$ in which the curve operates; A and B, the coefficients of the curve equation; $\mathtt{G}$, a generator of a subgroup of a prime order on the curve; n, the order of the subgroup that $\mathtt{G}$ generates; and h, the cofactor, which is equal to the number of curve points w divided by n. Elliptic curve groups are typically chosen to have small cofactors to limit the number of elements of small order on the curve and to limit the checks required to protect against small subgroup attacks [55]. NIST recommends a maximum cofactor for various curve sizes [239]. The NIST curves over prime order fields specified in FIPS 186-4 are in the Weierstrass form and have a cofactor 1, but curves in the Montgomery form always have a cofactor that is a multiple of 4 [227].

ECDH Encryption. We target the OpenPGP ECDH public-key encryption scheme, ECDH encryption, as specified in RFC 6637 [182] and defined as method `C(1e,1s,ECC CDH)` in NIST SP800-56A [49]. ECDH encryption is a hybrid scheme that combines elliptic curve Diffie-Hellman key exchange with a symmetric-key cipher such as AES. To generate a key pair given an elliptic curve group generator $\mathtt{G}$, Alice first generates a random scalar k as her private key, and computes $[k]\mathtt{G}$ as her public key. To encrypt a message m to Alice, Bob chooses a random scalar k' and computes $[k']([k]\mathtt{G})$, where $[k]\mathtt{G}$ is Alice's public key. Bob uses the result to derive a symmetric encryption key x. The message m is then symmetrically encrypted using x to obtain $\mathtt{Enc}_x(m)$, and the ciphertext is set to $c = (\mathtt{Enc}_x(m), \mathtt{P})$, where $\mathtt{P} = [k']\mathtt{G}$ is the ephemeral public key, which also plays the role of a ciphertext in our chosen ciphertext attack. To decrypt c, Alice computes $[k](\mathtt{P}) = [k]([k']\mathtt{G})$. She then derives from it a symmetric key x'. This key can then be used to symmetrically decrypt $\mathtt{Enc}_x(m)$ to get message m'. By the commutative property of elliptic curve scalar-by-point multiplication $[k]([k']\mathtt{G}) = [k']([k]\mathtt{G})$. Hence we have $x' = x$ and $m' = m$.

Point Representation. Elliptic curve points can be represented in many different forms. The canonical representation uses the *affine coordinates*, where a point on the curve is represented by a pair of integers (x, y) that satisfy the curve equation. However, this representation requires an expensive field inversion operation to add two elliptic curve points.

Using *projective coordinates*, where a point (x, y) is represented by the triplet (X, Y, Z), where $(x, y) = (X/Z, Y/Z)$ for $Z \neq 0$, obviates the field inversion [93]. A special "point at infinity" is represented by $Z = 0$. Points can have many different representations depending on the value of Z, and this equivalence class is denoted $(X : Y : Z)$.

Optimization for Montgomery Coordinates. Elliptic curve points support arithmetic operations based on the elliptic curve's group addition law. For Montgomery curves, the group addition law which adds two projective points (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) to produce the sum (X_s, Y_s, Z_s) computes X_s and Z_s without using the y-coordinates at all. This allows us to represent a point $\mathtt{P} = (x, y)$ without the y-coordinate using the *projective Montgomery coordinates* $\mathtt{P} = (X, Z)$, where $x = X/Z$ for $Z \neq 0$. This form loses some information: there is no way to distinguish between the points (x, y) and $(x, -y)$ since they both have the representation (X, Z), but this is not an issue for the application of ECDH key exchange. These x-coordinate point operations on Montgomery curves are extremely fast, and they also allow points to be represented with only half as many bits, so that a public key can be represented with only $x = X/Z$ instead of (x, y).

Low-Order Elements. Every elliptic curve group has an order-1 element called the identity element, which we will denote $\mathtt{G}_1$. $\mathtt{G}_1$ is often called the "point at infinity". For every prime divisor p_i of the group order w, there exists an element on the curve with order p_i. Because Montgomery curves must have a cofactor that is a multiple of 4, such curves must contain an element $\mathtt{G}_2$ of order 2. (That is because 2 is a prime that divides the group order). Next, since 4 divides the group order for Montgomery curves, there is also a subgroup of order 4. This does not imply that the curve has an order-4 element, but this is often the case. We denote order-4 elements as $\mathtt{G}_4$ when they exist. In the Montgomery projective coordinates, the point at infinity is represented by $(X \neq 0 : Z = 0)$, the element of order 2 by $(X = 0 : Z \neq 0)$. The coordinates of the elements of order 4, when they exist, depend on the specific curve.

Curve25519. Introduced by Bernstein [55], Curve25519 is specified in the Montgomery

form as $y^2 = x^3 + 486662x^2 + x$ over the field with prime modulus $p = 2^{255} - 19$. Curve25519 has a cofactor 8, meaning that the order of the curve is $8 \cdot n$, for a prime n. Curve25519 also has two order-4 elements with affine coordinates $(x = 1, y = \pm\sqrt{486664})$. Both these elements are represented in the Montgomery projective coordinates by $(X = \lambda : Z = \lambda)$, where $\lambda \neq 0$. The curve has no element with affine x-coordinate $x = -1$, however such elements, represented by $(X = \lambda : Z = -\lambda)$ exist on the twist of the curve, where they have an order 4. For the purposes of this work, the elements of order 4 on the curve and on the curve's twist behave in a similar manner and we refer to all of them as $\mathtt{G}_4$.

When introduced, Curve25519 timings were more than twice as fast as previously reported times for elliptic curves of an equivalent security level, while also including "free key compression, free key validation, and state-of-the-art timing-attack protection" [55]. Implementations are not required to perform key validation, since by definition secret keys have the low-order bits set to zero, so there is no risk of leaking these bits in a small subgroup attack [55]. Moreover, the use of the Montgomery ladder scalar multiplication algorithm provides side-channel resistance [184, 243]. Curve25519 was standardized by RFC 7748 [200], and is implemented in a wide variety of protocols and software [173].

Public Key Validation for Curve25519. Part of the appeal of using Diffie-Hellman with Curve25519 is that implementations are not required to validate public keys, including the ephemeral public key in ECDH. Not only is validation not required, but the recommendation is to not validate public keys because "The Curve25519 function was carefully designed to allow all 32-byte strings as Diffie-Hellman public keys" [56]. This recommendation is the subject of debate, where proponents claim that key validation is not required [257] whereas critics maintain that the recommendation is risky [42, 111].

In this work we identify another risk associated with this recommendation. The recommendation implicitly assumes that the implementations of the curve functions and of the underlying field arithmetic are constant-time. Our attack exploits the failure to reject low-order elements, combined with a non-constant-time implementation of the underlying field

arithmetic.

4.2.2. Scalar-by-Point Multiplication

Scalar-by-point multiplication is one of the core operations in elliptic curve cryptography. Given a positive scalar k and an elliptic-curve point $\mathtt{P}$, the scalar-by-point multiplication operation adds $\mathtt{P}$ to itself k times to produce the point $[k]\mathtt{P}$. There are several popular methods for implementing scalar-by-point multiplication in the literature.

Double-And-Add. The simplest method is the double-and-add method, which is similar to the square-and-multiply algorithm in modular exponentiation. For each bit of the scalar k, the algorithm performs one doubling operation. Additionally, in case the bit is set, the algorithm also performs an addition operation. However, the fact that the sequence of doubles and adds performed by this algorithm leaks the bits of k is a major side channel weakness [99].

Montgomery Ladder. Implementations that wish to protect against side channel attacks can use the Montgomery ladder algorithm [227] for scalar-by-point multiplication. This algorithm performs the same number of addition and double operations regardless of the value of the scalar k. As such, the algorithm can be implemented without any key-dependent branches, making it more side channel resistant [184, 243].

The Montgomery ladder is based on the observation that given $[\lfloor n/2 \rfloor]\mathtt{P}$ and $[\lfloor n/2 \rfloor + 1]\mathtt{P}$, we can easily calculate $[n]\mathtt{P}$ and $[n+1]\mathtt{P}$. More specifically, if we have $\mathtt{R}_0 = [\lfloor n/2 \rfloor]\mathtt{P}]$ and $\mathtt{R}_1 = [\lfloor n/2 \rfloor + 1]\mathtt{P}$, for even n we calculate $\mathtt{R}_1 \leftarrow \mathtt{R}_0 + \mathtt{R}_1, \mathtt{R}_0 \leftarrow [2]\mathtt{R}_0$, and for odd n we use $\mathtt{R}_0 \leftarrow \mathtt{R}_0 + \mathtt{R}_1, \mathtt{R}_1 \leftarrow [2]\mathtt{R}_1$. We note that in both cases we perform one addition and one doubling operation and the only difference between the cases is the roles that the variables play.

Naive implementations of the Montgomery ladder scan the scalar from the most significant bit to the least significant. For each bit, they conditionally execute one of the computations specified above, based on the value of the bit. However, such implementations are known

Algorithm 1 Montgomery ladder scalar-by-point multiplication operation.

Input: A positive scalar k and an elliptic-curve point $\mathtt{P}$, where $k = \sum_{i=0}^{n-1} 2^i \cdot k_i$ and $k_i \in \{0, 1\}$ for all $i = 0, \cdots, n-1$.
Output: $[k]\mathtt{P}$.

1: **procedure** MONTGOMERY_LADDER($k, \mathtt{P}$)
2: $\quad \mathtt{R}_0 \leftarrow \mathtt{G}_1$ ▷ $\mathtt{G}_1$ represents the order-1 identity element
3: $\quad \mathtt{R}_1 \leftarrow \mathtt{P}$
4: $\quad \mathtt{dif\_x} \leftarrow \mathtt{P}.x$
5: $\quad$ **for** $i \leftarrow n-1$ **to** 0 **do**
6: $\quad\quad b \leftarrow k_i$
7: $\quad\quad \mathtt{Q}_0, \mathtt{Q}_1 \leftarrow$ CONDITIONAL_SWAP($\mathtt{R}_0, \mathtt{R}_1, b$) ▷ Constant time swap when $b = 1$
8: $\quad\quad \mathtt{S}_0, \mathtt{S}_1 \leftarrow$ MONTGOMERY_STEP($\mathtt{Q}_0, \mathtt{Q}_1, \mathtt{dif\_x}$) ▷ $\mathtt{S}_0 = [2]\mathtt{Q}_0, \mathtt{S}_1 = \mathtt{Q}_0 + \mathtt{Q}_1$
9: $\quad\quad \mathtt{R}_0, \mathtt{R}_1 \leftarrow$ CONDITIONAL_SWAP($\mathtt{S}_0, \mathtt{S}_1, b$) ▷ Constant time swap when $b = 1$
10: $\quad$ **return** $\mathtt{R}_0$

to be vulnerable to side channel attacks [28, 307]. A common mitigation, which Libgcrypt uses, is to conditionally swap the values of $\mathtt{R}_0$ and $\mathtt{R}_1$ before and after the computation. Algorithm 1 shows the pseudocode of such an implementation. The conditional swaps can be implemented using bit manipulations to avoid any branches or memory access operations that depend on secret-key bits. Such implementations are protected against timing and cache-based side channel attacks.

As mentioned earlier, one of the advantages of Montgomery curves is that the *Montgomery step*, which sums its two arguments and doubles one of them (Line 8 of Algorithm 1), can be calculated efficiently using only the x-coordinates in the projective Montgomery form. Algorithm 2 shows a pseudo code of an implementation of the Montgomery step. We note that the implementation does not contain any branches or memory accesses that depend on secret values.

4.2.3. Libgcrypt's Implementation

We now describe Libgcrypt's implementation of Montgomery curves and point operations. Libgcrypt stores points using projective Montgomery coordinates. Each point is represented as a pair (X, Z), where each element is a large integer stored using Libgcrypt's arithmetic library, MPI. MPI stores large integers as arrays of *limbs*, which are 64-bit words on the x86-64 architecture used in our tests. For Curve25519, field elements are calculated modulo

Algorithm 2 Libgcrypt's Montgomery step operation (simplified).

Input: Two points $\mathtt{Q}_0 = (X_0, Z_0)$ and $\mathtt{Q}_1 = (X_1, Z_1)$ in projective coordinates on an elliptic-curve based group of order p, and `dif_x` which should be equal to the difference in x-coordinates of the input points.

Output: Two points $\mathtt{Dbl} = (X_d, Z_d)$ and $\mathtt{Sum} = (X_s, Z_s)$ in projective coordinates such that $\mathtt{Dbl} = [2]\mathtt{Q}_0$ and $\mathtt{Sum} = \mathtt{Q}_0 + \mathtt{Q}_1$.

1: **procedure** MONTGOMERY_STEP($\mathtt{Q}_0, \mathtt{Q}_1, \mathtt{dif\_x}$)
2: $l_1 \leftarrow X_1 + Z_1 \bmod p$
3: $l_2 \leftarrow X_1 - Z_1 \bmod p$
4: $l_3 \leftarrow X_0 + Z_0 \bmod p$
5: $l_4 \leftarrow X_0 - Z_0 \bmod p$
6: $l_5 \leftarrow l_4 l_1 \bmod p$
7: $l_6 \leftarrow l_3 l_2 \bmod p$
8: $l_7 \leftarrow l_3^2 \bmod p$
9: $l_8 \leftarrow l_4^2 \bmod p$
10: $l_9 \leftarrow l_5 + l_6 \bmod p$
11: $l_{10} \leftarrow l_5 - l_6 \bmod p$
12: $X_d \leftarrow l_7 l_8 \bmod p$
13: $l_{11} \leftarrow l_7 - l_8 \bmod p$ ▷ $l_{11} = 4X_0Z_0$ (see Equation 4.5)
14: $X_s \leftarrow l_9^2 \bmod p$
15: $l_{12} \leftarrow l_{10}^2 \bmod p$
16: $l_{13} \leftarrow l_{11} \cdot (\mathtt{A} - 2)/4 \bmod p$ ▷ $\mathtt{A} = 486662$ for Curve25519
17: $Z_s \leftarrow l_{12} \cdot \mathtt{dif\_x} \bmod p$
18: $l_{14} \leftarrow l_7 + l_{13} \bmod p$
19: $Z_d \leftarrow l_{14} l_{11} \bmod p$
20: **return** $((X_d, Z_d), (X_s, Z_s))$

$2^{255} - 19$ hence integers can have up to four limbs. Multiplication and squaring operations on field elements can be up to 510 bits long before modular reduction and may require 8 limbs for storage.

Libgcrypt's Scalar-by-Point Multiplication. Libgcrypt uses the Montgomery ladder (Algorithm 1) for scalar-by-point multiplication. In order to protect from side channel attacks, Libgcrypt's implementation uses a side-channel-resistant constant-time point swap function to set the inputs and outputs of the MONTGOMERY_STEP function based on the value of the secret key bit in each loop iteration.

Libgcrypt's Montgomery Step Implementation. The MONTGOMERY_STEP function receives inputs $\mathtt{Q}_0$, $\mathtt{Q}_1$, and `dif_x` which is the affine x-coordinates of the input point `P`. It

returns ([2]$\mathtt{Q}_0, \mathtt{Q}_0$+$\mathtt{Q}_1$). Doubling of $\mathtt{Q}_0$, represented in the projected Montgomery coordinates as (X_0, Z_0), is computed by

$$X_d = (X_0 + Z_0)^2(X_0 - Z_0)^2 \tag{4.1}$$

$$Z_d = (4X_0Z_0)((X_0 + Z_0)^2 + ((\mathtt{A} - 2)/4) * (4X_0Z_0)), \tag{4.2}$$

and the Montgomery addition operation for computing $\mathtt{Q}_0 + \mathtt{Q}_1$ performs

$$X_s = ((X_0 - Z_0)(X_1 + Z_1) + (X_0 + Z_0)(X_1 - Z_1))^2 \tag{4.3}$$

$$Z_s = \mathtt{dif\_x}((X_0 - Z_0)(X_1 + Z_1) - (X_0 + Z_0)(X_1 - Z_1))^2, \tag{4.4}$$

where $\mathtt{A}$ is a curve parameter.

Algorithm 2 shows a simplified version of Libgcrypt's implementation of the MONTGOMERY_STEP algorithm for projective Montgomery coordinates. The actual Libgcrypt implementation reuses the coordinates of the input variables for temporary storage during the computation and precomputes $(A - 2)/4$. For clarity, we replace these with local variables and explicit formulas.

We pay special attention to the multiplication on Line 19, which we target in Section 4.3. In particular we note that the value l_{11} computed in Line 13 of Algorithm 2 is

$$\begin{aligned} l_{11} &= l_7 - l_8 = l_3^2 - l_4^2 \\ &= (X_0 + Z_0)^2 - (X_0 - Z_0)^2 \\ &= (X_0^2 + 2X_0Z_0 + Z_0^2) - (X_0^2 - 2X_0Z_0 + Z_0^2) \\ &= 4X_0Z_0. \end{aligned} \tag{4.5}$$

Libgcrypt's Modular Reduction Routine. After each arithmetic operation in MONTGOMERY_STEP (Algorithm 2), the result is reduced modulo p using Libgcrypt's modular reduction function. Algorithm 3 shows a simplified version of this function, which uses the

Algorithm 3 Libgcrypt's modular reduction operation (simplified).

Input: Two integers x and m, represented as a sequence of limbs $x_0 \dots x_{l-1}$ and $m_0 \dots m_{k-1}$.
Output: $x \bmod m$.

1: **procedure** MODULAR_REDUCTION(x, m)
2: $l \leftarrow$ SIZE_IN_LIMBS(X)
3: $k \leftarrow$ SIZE_IN_LIMBS(M)
4: **if** $l < k$ **then**
5: **return** x ▷ Early exit if x is smaller than m
6: **for** $i \leftarrow l - 1$ **downto** $k - 1$ **do**
7: $q \leftarrow (x_i \cdot 2^{64} + x_{i-1})/m_{k-1}$ ▷ Estimate quotient q
8: **if** $q(m_{k-1} \cdot 2^{128} + m_{k-2}) > x_i \cdot 2^{128} + x_{i-1} \cdot 2^{64} + x_{i-2}$ **then**
9: $q \leftarrow q - 1$ ▷ If q is too large, adjust estimate
10: $x \leftarrow x - q \cdot m \cdot 2^{64(i-k)}$ ▷ Subtract from x
11: **return** x ▷ x holds the remainder

classical long division algorithm formalized by Knuth [198]. The quotient q is estimated in each iteration of the loop and adjusted if the initial estimate was off by 1. Then, the appropriate multiple of q is subtracted from the input before execution returns to the top of the loop. Notice that code execution only reaches the body of the main `for` loop at Line 6 when the number of limbs of the number being reduced, is equal to or greater than the number of limbs of m, the modulus. Otherwise, when the input is shorter, and therefore guaranteed to be smaller, than m, the algorithm exits early without performing a modular reduction.

As we show in Section 4.3, detecting the early exit in Line 5 shows that the value $l_{14} \cdot l_{11}$, as computed in Line 19 of Algorithm 2, is smaller than the order of the group, p, allowing the attacker to determine the order of the group elements being multiplied. Using this information, the attacker can then extract the bits of the secret scalar k, resulting in a complete key extraction.

4.3. Cryptanalysis

In this section we present our non-adaptive chosen ciphertext side-channel attack against Libgcrypt's ECDH implementation. Since the sequence of arithmetic field operations performed by the Montgomery ladder is not key-dependent, we wish to find some elliptic curve

point P that, when multiplied by the secret key k, will cause an observable correlation between the intermediate values used as *operands* of these arithmetic operations and the bits of k. We then use a side-channel attack to obtain information about the values of the operands of these operations, achieving complete key recovery.

Chosen Ciphertext as Order-2 Element. Previous work [139, 309] used an order-2 element as a chosen ciphertext for attacks on RSA and ElGamal in order to create an observable correlation between the operands of the arithmetic operations performed by the exponentiation routine and the secret key. Unfortunately, this approach does not work in our case. The order 2 element is $\mathtt{G}_2 = (X = 0, Z \neq 0)$. If we use $\mathtt{P} = \mathtt{G}_2$, we have $\mathtt{dif\_x} = \mathtt{G}_2.x = 0$ in Line 4 of Algorithm 1. As Ransom [267] observes, this is an exceptional case that causes incorrect results for the Montgomery addition computed by MONTGOMERY_STEP. More specifically, because Z_s is set to 0 on Line 17 of Algorithm 2, the sum $(X_s, Z_s) = \mathtt{G}_1 + \mathtt{G}_2$ is computed as $(X = 0, Z = 0)$, which is illegal in the Montgomery projective representation. Subsequent iterations of the loop in Algorithm 1 treat this undefined point as $\mathtt{G}_1$ instead of $\mathtt{G}_2$. The consequence of this irregularity is that when we use $\mathtt{P} = \mathtt{G}_2$, all of the intermediate values in Algorithm 1 are the invalid point irrespective of the secret key bits. We stress that the irregularity in the implementation only happens when $\mathtt{P} = \mathtt{G}_2$. For every other value of P, the point addition will involve at least one value that is neither $\mathtt{G}_1$ nor $\mathtt{G}_2$ and the results of the algorithm are correct.

4.3.1. Long and Short Modular Reductions and Order-2 Elements

Our attack exploits the early exit in Line 5 of Algorithm 3. We say that the modular reduction in $l_{14} \cdot l_{11} \bmod p$ (Line 19 of Algorithm 2) is short when the number of limbs in $l_{14} \cdot l_{11}$ is smaller than the number of limbs in p, causing an early exit. Otherwise, we say that modular reduction in $l_{14} \cdot l_{11} \bmod p$ is long. We later show that by monitoring the cache, we can detect the early exit. We now proceed to describe when early exits occur and how we can recover the key based on them.

Order-1 and Order-2 Arguments Imply Short Modular Reductions. Consider

the case where the first argument $\mathtt{Q}_0$ to MONTGOMERY_STEP (Algorithm 2) is either the order-1 element $\mathtt{G}_1$ or the order-2 element $\mathtt{G}_2$. As mentioned in Section 4.2.1, for $\mathtt{G}_1$ we have $(X_0 \neq 0, Z_0 = 0)$ and for $\mathtt{G}_2$ we have $(X_0 = 0, Z_0 \neq 0)$. In both cases the value $l_{11} = 4X_0Z_0$ (see Equation 4.5) computed in Line 13 is equal to 0. Next, since l_{11} is zero we obtain that the value $l_{14} \cdot l_{11}$ computed in Line 19 is also equal to 0. Finally, since the representation of 0 consists of only one limb, the condition in line Line 4 of Algorithm 3 is true, causing an early exit on Line 5, and the modular reduction in $Z_d \leftarrow l_{14} \cdot l_{11} \bmod p$ is short.

Order-4 Arguments Typically Imply Long Modular Reductions. As we discuss in Section 4.2.1, an order-4 element $\mathtt{G}_4$ has the form $(X = \lambda, Z = \pm\lambda)$, with $\lambda \in [1, \ldots, p-1]$. The fact that the affine point $x = 1$ can be expressed in this way with projective coordinates actually helps our attack. As above, consider passing the order-4 element $(X_0 = \lambda, Z_0 = \pm\lambda)$ as the $\mathtt{Q}_0$ argument of MONTGOMERY_STEP. We now look at the values of l_{11} and l_{14} used in Line 19. From Equation 4.5 we have $l_{11} = 4X_0Z_0 = \pm 4\lambda^2$.

For l_{14} we have:

$$\begin{aligned} l_{14} &= l_7 + l_{13} \bmod p = l_3^2 + l_{11} \cdot (A-2)/4 \bmod p \\ &= (X_0 + Z_0)^2 + 4\lambda^2 \cdot (A-2)/4 \bmod p \\ &= \lambda^2 \cdot (A \pm 2) \bmod p \end{aligned}$$

where the ± 2 depends on whether G_4 is on the curve or on its twist, i.e. whether $Z_0 = \lambda$ or $Z_0 = -\lambda$. Consequently, if $\lambda < (2^{192}/(A+2))^{1/4}$ or $p - \lambda < (2^{192}/(A+2))^{1/4}$, we have that $l_{14}l_{11} < 2^{192}$ and the reduction in Line 19 is short. Otherwise, we have that $l_{14}l_{11} > 2^{192}$ and the reduction is long, except with a negligible probability of $2^{192-510}$.

4.3.2. Order-4 Element as a Chosen Ciphertext

We now consider decryption when the adversary sends an element of order 4 $\mathtt{G}_4$ as chosen ciphertext. Recall that there are two elements of order 4, an element on the curve, with affine x-coordinate of 1 and an element on the twist with x-coordinate of -1. However, for our purposes these elements behave the same so we refer to both as $\mathtt{G}_4$. The relevant rules

of point addition for order-4 elements are as follows:

$$[2]\mathtt{G}_4 = \mathtt{G}_2$$

$$\mathtt{G}_1 + \mathtt{G}_4 = \mathtt{G}_4$$

$$\mathtt{G}_2 + \mathtt{G}_4 = \mathtt{G}_4$$

Montgomery Ladder Invariant Revisited. Next, we recall that in the Montgomery ladder, the difference in affine coordinates of the tracked values $\mathtt{R}_0$ and $\mathtt{R}_1$ is $\mathtt{P}$, the input point. Based on the addition rules above, when the input point is $\mathtt{G}_4$, as is the case in our attack, one of $\mathtt{R}_0$ and $\mathtt{R}_1$ must be $\mathtt{G}_4$ and the other must be either $\mathtt{G}_1$ or $\mathtt{G}_2$.

Determining Key Bits. We now show how, an attacker that knows the value of the i-th key bit, k_i can leverage the side channel leakage to learn the value of bit k_{i-1}. Repeating this argument for all of the bits of k results in a complete key extraction. Indeed, note that based on the invariant and the rules above, every time the MONTGOMERY_STEP function is executed in Algorithm 1, the output value $\mathtt{S}_1 = \mathtt{Q}_0 + \mathtt{Q}_1$ must be an order 4 element $\mathtt{G}_4$. Next, since $\mathtt{S}_1 = \mathtt{G}_4$ the Montgomery ladder invariant implies that $\mathtt{S}_0$ is either $\mathtt{G}_1$ or $\mathtt{G}_2$. The values held by $\mathtt{S}_0$ and $\mathtt{S}_1$ after processing bit k_i will propagate to the Montgomery step of bit k_{i-1} as the values held by $\mathtt{Q}_0$ and $\mathtt{Q}_1$, possibly getting swapped at two locations: Line 9 if bit k_i is set, and Line 7 in the next loop iteration in case bit k_{i-1} is set.

Thus, we consider the following two cases based on the values of the key bits k_i and k_{i-1}:

1. $\boldsymbol{k_{i-1} = k_i}$**.** When propagating from $\mathtt{S}_0$ and $\mathtt{S}_1$ to $\mathtt{Q}_0$ and $\mathtt{Q}_1$, the values will either be swapped twice if $k_i = k_{i-1} = 1$, or not swapped at all, when $k_i = k_{i-1} = 0$. In both cases, $\mathtt{Q}_0 \in \{\mathtt{G}_1, \mathtt{G}_2\}$ and $\mathtt{Q}_1 = \mathtt{G}_4$. As stated in Section 4.3.1, having $\mathtt{Q}_0 \in \{\mathtt{G}_1, \mathtt{G}_2\}$ implies that the modular reduction in Line 19 of Algorithm 2 performed during the processing of k_{i-1} will be short.

2. $\boldsymbol{k_{i-1} \neq k_i}$**.** When propagating from $\mathtt{S}_0$ and $\mathtt{S}_1$ to $\mathtt{Q}_0$ and $\mathtt{Q}_1$, the values will be swapped

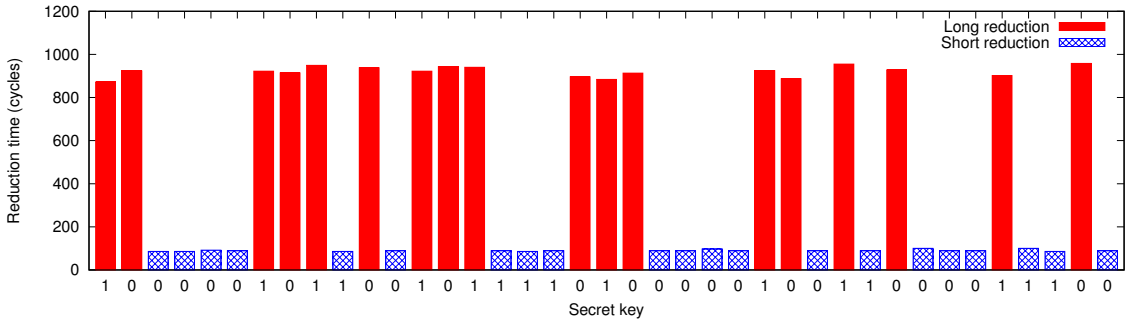


Figure 2: **Trace (excluding four first bits) of scalar-by-point multiplication of a secret key with an element of order 4**—We can learn the bits of the scalar (shown on the x-axis) from the sequence of long and short modular reduction operations: a short reduction implies that the current bit is the same as the previous bit, whereas a long reduction means that the current bit is the complement of the previous bit.

exactly once, since only one of k_i and k_{i-1} is set. In either case, $\mathtt{Q}_0 = \mathtt{G}_4$ and $\mathtt{Q}_1 \in \{\mathtt{G}_1, \mathtt{G}_2\}$. As stated in Section 4.3.1, having $\mathtt{Q}_0 = \mathtt{G}_4$ implies that the modular reduction in Line 19 of Algorithm 2 performed during the processing of k_{i-1} will be long.

Hence, when the attacker knows k_i, observing the length of the modular reduction will allow the attacker to determine the value of k_{i-1}. This culminates in an easy procedure for recovering bits directly from a sequence of short and long reductions: a short reduction means that the current bit is the same as the previous bit, and a long reduction means that the current bit is the complement of the previous bit.

Key Extraction. Confirming the above, in Figure 2 we show a sequence of modular reductions performed in Line 19 during 39 loop iterations of Montgomery ladder (Algorithm 1). As can be seen, some modular reductions are long while others are short, which clearly indicates the leakage of secret key material.

Assuming that the bit preceding the captured sequence was 0, we apply our easy rule: a long reduction implies that the value of the next bit (the first captured) is 1. The next modular reduction is long again, and we can conclude that the bit is 0. The third reduction is short, indicating that the value of the bit remains 0 and so forth.

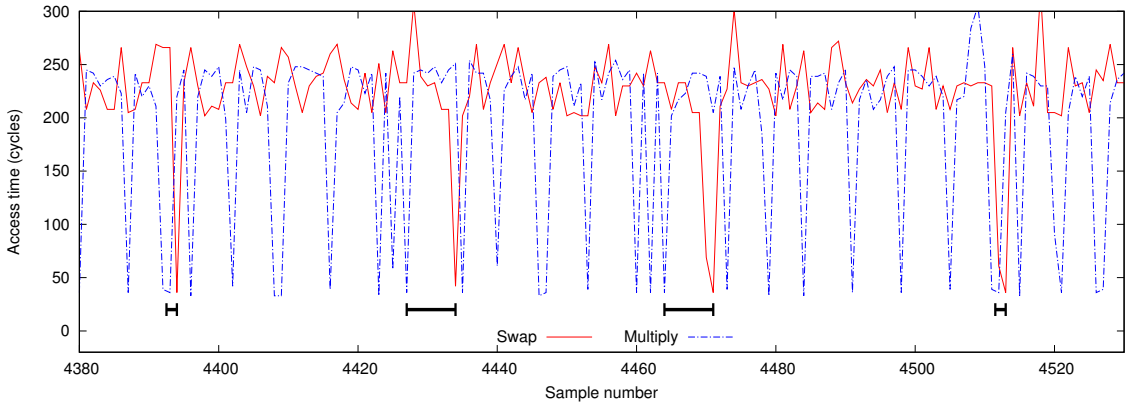


Figure 3: **Memory access times of the Flush+Reload attack**–The lengths of the horizontal bars corresponding to the lengths of modular reductions. The results were obtained by flushing and reloading four memory locations, two within the constant-time swap code and two within the multiplication code. In each sample, we perform a flush followed by a reload for each of these four memory locations, measuring access times. We show the minimum of the access times for the two memory locations in the constant-time swap code in red, and the minimum of the access times for the two memory locations in the multiplication code in blue.

Small values of λ. A minor limitation of the above approach is that, as discussed above, when doubling $\mathtt{G}_4$ with a small λ, the modular reduction will be short. Experimentally, we find that during most of the algorithm the probability of this happening is negligible. However, when Libgcrypt initializes $\mathtt{R}_1$, it sets $\lambda = 1$. Nevertheless, the length of λ increases rapidly, reaching the full size of four limbs (255 bits) within four loop iterations. However, during these first four iterations the value of λ is small, hence our attack is unable to determine the first four key bits used during these iterations.

4.4. Experimental Results

4.4.1. Attack Technique

For the side channel, we use the Flush+Reload attack [308] in conjunction with the amplification attack of Allan et al. [34]. Microarchitectural attacks such as Flush+Reload leak information on programs by monitoring the effects that executing a program has on the state of the components of the processor. See Ge et al. [136] for a survey of published microarchitectural attacks. In particular, the Flush+Reload attack leaks information by monitoring the presence of memory locations in the cache.

The Flush+Reload Attack. The Flush+Reload attack consists of two phases. In the *flush* phase, the attacker evicts the contents of one or more monitored memory addresses from the cache. This is typically achieved by using a dedicated instruction, such as the x86 `clflush`, but in the absence of such an instruction, the attacker can use other mechanisms to achieve eviction [157, 315]. After the flush phase is completed the attacker waits for a short while to allow the victim time to execute. Then, during the *reload* phase, the attacker reads the contents of the memory addresses, measuring the time it takes to perform the read.

In case the victim accesses one or more of the monitored memory addresses between the flush and the reload phases, the contents of these addresses will be cached again causing the attacker's reads to be fast. Conversely, in case the victim does not access a monitored memory address, the contents will not be cached, causing the attacker's read to take longer. Performing the attack repeatedly, the attacker can trace the victim's memory accesses to specific addresses over time. In case the monitored memory addresses are part of the victim's code, the attacker learns some information about the victim's execution patterns.

The Amplification Attack. Because the Flush+Reload attack executes concurrently with the victim, the Flush+Reload attack has a limited temporal resolution. To improve the attack resolution, Allan et al. [34] suggest slowing the victim down. At high level, this is done by identifying frequently accessed, or "hot", sections of the victim code and then repeatedly evicting these sections from the cache. Next, in order to execute code that has been evicted, the victim has to wait until the processor loads the code from the main memory. This, in turn, increases the time it takes the victim to execute each operation and provides a larger time window for the attacker to make accurate side-channel measurements. To evict the code from the cache, Allan et al. [34] use the `clflush` instruction, hence like the Flush+Reload attack, amplification only works when the victim and the attacker share memory.

4.4.2. Attacking the Scalar-by-Point Multiplication

Experimental Setup. We target Libgcrypt's implementation of the Montgomery ladder scalar-by-point multiplication routine. We first demonstrate the attack's feasibility by directly invoking Libgcrypt's scalar multiplication on an order-4 element. As described in Section 4.1.3, we target Libgcrypt 1.7.6, which is the latest version of Libgcrypt at the time of writing this paper, as supplied in the latest Ubuntu 17.04. Below, all experiments and cache attacks were performed on a Dell Optiplex 9010 desktop, equipped with an i7-3770 3.4 GHz processor and 8GB of memory, running unmodified Ubuntu 17.04. To mount the Flush+Reload attack, we used the `FR-trace` utility of the Mastik toolkit [306]. `FR-trace` provides a command-line interface for performing the Flush+Reload attacks as well as support for the amplification attack of Allan et al. [34].

Applying the Flush+Reload Attack. To extract information about whether the modular reduction in Line 19 of Algorithm 2 was long or short during each iteration of the main loop of Algorithm 1, we set `FR-trace` to monitor four memory locations within the Libgcrypt library. Two of these locations are within the field multiplication code (which executes before the modular reduction operation) and the other two are within the CONDITIONAL_SWAP function (which executes after the modular reduction operation). As Allan et al. [34] observe, monitoring two memory locations with the same functionality reduces the probability that the attack will miss a memory access due to overlap between the victim's memory accesses during the attacker's reload phase. To improve our ability to detect the length of the modular reduction operation, we use the amplification attack of [34] to repeatedly evict the code of the operation. This increases the time to perform modular reduction by a multiplicative factor of 11.1.

Recall that our attack correlates the bits of the secret key and the time it takes to perform the modular reduction in Line 19 of Algorithm 2. Since this modular reduction operation is executed between our two measurement points, we expect that the temporal separation between the two measurements will reveal the length of the modular reduction, i.e. whether

it is long or short.

Trace Analysis. Figure 3 shows a sample of a trace of a scalar multiplication. For each measured functionality (field multiplication code and the CONDITIONAL_SWAP function) we plot the shorter of reload times of the two measurement locations. Recall that the reload time of a monitored location is shorter following a victim's access to that location. In our test environment, we find that loads from memory take over 150 cycles, whereas loads from the cache take less than 100 cycles. Thus, whenever the reload takes below 100 cycles we can assume that the victim has accessed the monitored location.

Observing Swap Operations. Looking at Figure 3, we see a sequence of "dips" which indicate various victim accesses. Dips in the swap line (solid red) indicate that the victim performed the constant time swap operation. Due to the low temporal resolution of the Flush+Reload attack, we are unable to distinguish between the swap that occurs at the end of one loop iteration of Algorithm 1 and the swap at the start of the next one. Hence, the four dips visible in the solid red line show the times where processing of one scalar bit ends and processing of the following bit starts during the main loop of Algorithm 1.

Observing Multiplication Operations. Dips in the multiply line (dashed blue) indicate times when the victim performed the multiplication operations in Algorithm 2. Gaps between the dips correspond to all of the other operations that the algorithm performs. Due to the amplification attacks, the dominant component in the gaps is the time it takes to compute the modular reduction.

The amplification attack only amplifies the main loop of the modular reduction. Hence, when Algorithm 3 exits early, its timing is not affected by the attack. Due to the limited temporal resolution of the Flush+Reload attack, in the case of a short reduction, the attack is unable to distinguish between the timing of the multiplication in Line 19 of Algorithm 2 and the following swap operation.

Observing Long and Short Modular Reductions. We now turn our attention to the

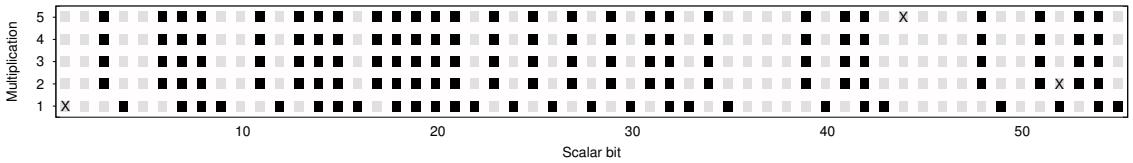


Figure 4: **Five processed traces**—Dark spots indicate an observed long reduction and light spots indicate an observed short reduction. Three errors in the observation are marked with X marks. Two of them observe the wrong reduction length and the third is a superfluous bit.

gap between the last *observed* multiplication operation and the following swap. These are marked with black horizontal bars. We note that in the case of a long reduction this gap is due to the modular reduction in Line 19 of Algorithm 2. However, as discussed above, in the case of a short reduction, Flush+Reload samples this multiplication in the same time as the swap operation. Hence, the gap is due to the preceding multiplication, in Line 16. Because one of the multiplicands in Line 16 is short, the multiplication result is short and the modular reduction in this case is faster than that of a long reduction.

As we can see, Figure 3 shows one short gap, followed by two long and another short gap. These correspond to long and short modular reductions. Hence, by measuring the length of the gap, the attacker can recover the information on the length of the last modular reduction, and from it recover the bits of the key.

Handling Measurement Errors. Side-channel attacks rarely produce error-free results. To measure the number of errors in our attack, we captured 1000 traces and compared with the ground truth. On average, there are 3.8 errors in a trace. See Figure 5 for the distribution of the number of errors in traces.

Overall Attack Performance. To correct the errors, we selected five arbitrary traces (see Figure 4), aligned them manually (about 10 minutes of wall-clock time) and used a simple majority rule to decide the length of each modular reduction operation. From this we were able to deduce for all but the leading four key bits whether the modular reduction in Line 19 of Algorithm 2 was long or short. Finally, applying the cryptanalysis from Section 4.3, we

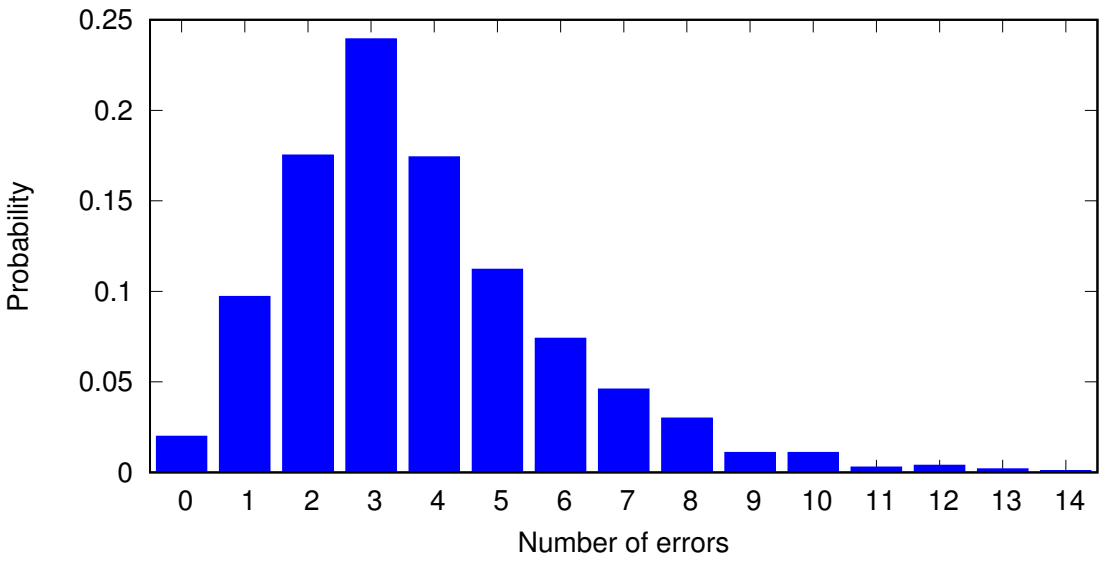


Figure 5: **Distribution of the number of errors (excluding four first bits) in traces of the scalar multiplication**—Out of 1000 captured traces, there are an average of 3.8 errors per trace.

successfully recovered all but the first four bits of a randomly generated Curve25519 scalar. The leading bits can then easily be found using exhaustive search.

4.4.3. Attacking Applications

We now turn our attention to attacking applications that use Libgcrypt. We attack three applications: git-crypt [44], Pidgin's OpenPGP plugin [18, 149], and Enigmail [266]. We first describe these applications with a focus on how they use encryption and the attack vector. We then describe the attack results.

Git-crypt. Git-crypt is a plugin for the git revision control system, used to selectively encrypt files in a repository. When initialized, git-crypt selects a random AES key, which is used for encrypting the files stored in the git repository. To publish the repository's AES key, git-crypt creates encrypted key files using the Gnu Privacy Guard (GnuPG) software. Each of the key files is encrypted with the public key of an authorized user and is stored in the repository. When git processes modifications to an encrypted file, it invokes git-crypt, which calls GnuPG to retrieve the repository's AES key. Git-crypt then encrypts or decrypts the modified file.

We use the default install of git-crypt on Ubuntu 17.04. To attack, we modify the victim's encrypted key file by replacing the ECDH ephemeral public key with the element of order 4 and commit the change into the repository. Once the victim pulls the modified key file, any attempt to encrypt or decrypt files in the repository will send an element of order 4 into Libgcrypt's scalar multiplication routine, allowing the attacker to collect side channel information.

Running the attack on real-world software rather than on the scalar multiplication code only, presents two problems. The first is that GnuPG performs several public key operations when trying to match the public key used for encrypting the key file with the victim's key storage (called *keyring* in the GnuPG nomenclature). These operations access both the constant-time swap code and the multiplication code which our attack monitors. Consequently, the side channel attack collects much more information and we need to distinguish between the ECDH scalar multiplication operation and the other operations. To achieve that, we also use `FR-trace` to monitor the entry to the ECDH decryption code and ignore all accesses to monitored code that precede the entry.

The second problem we witness is that when running more software the system is more noisy, increasing the error rate. On average, we find that we have 14.9 errors in a trace and therefore we require 11 traces to recover the secret key.

Pidgin. Pidgin is a popular open-source chat application that supports communication across a variety of chat networks [18]. We target Pidgin's OpenPGP plugin [149], which allows a sender to encrypt messages with the recipient's public GnuPG key. When the recipient has the plugin enabled and receives a PGP-encrypted message, the message is automatically decrypted using GnuPG with no action required by the recipient.

We use the default APT distribution of Pidgin and the OpenPGP plugin for Ubuntu 17.04. To carry out the attack, we first enable PGP for the chat session and then send a chat message, replacing the ECDH ephemeral public key with an element of order 4. When the

victim receives the message, Pidgin uses GnuPG to decrypt the ciphertext, calling the scalar multiplication function in Libgcrypt with the order-4 element and enabling the side-channel attack.

We sent 100 malicious Pidgin messages containing an order-4 element to the target machine, while monitoring its cache activity. This resulted in 100 traces containing an average of 7.6 errors with 3 of the traces containing unusable data. Overall we recovered the victim key using information from 7 traces.

Enigmail. Enigmail is an add-on for the Mozilla Thunderbird email client that enables the sender to encrypt emails using the recipient's public GnuPG key. When the recipient views a GnuPG-encrypted email, Enigmail passes the ciphertext to GnuPG to be decrypted.

For our attack, we assume that the victim is running Mozilla Thunderbird in Ubuntu 17.04 with the default version of Enigmail installed. The attacker sends a GnuPG-encrypted email with the ECDH public key replaced with an order-4 element. When the victim clicks on the encrypted email, Enigmail passes the ciphertext to GnuPG for decryption, enabling a side-channel attack similar to the above.

Similar to the Pidgin attack above, we used Enigmail to decrypt 100 encrypted email messages containing order-4 elements on the target machine while monitoring its cache activity. This resulted in 100 traces containing an average of 9.1 errors with 9 of the traces containing unusable data. Overall we recovered the victim key using information from 7 traces.

4.5. Software Countermeasures

Our attack works by passing specially chosen ciphertexts (order-4 curve points) to the ECDH decryption routine to be multiplied by the secret scalar. Due to the mathematical structure of these inputs and the Montgomery ladder algorithm, they trigger key-dependent leakage patterns deep inside Libgcrypt's basic finite field arithmetic operations. Observing these patterns using the cache side channel, we are able to recover the secret key. We now

briefly review common countermeasures for preventing such chosen ciphertext attacks. See Fan et al. [124] and Fan and Verbauwhede [123] for more extended discussions.

Constant Time Arithmetic. Both the original publication of Curve25519 [55] and the NaCl library [59] use constant-time field arithmetic. Replacing Libgcrypt's code with any of these implementations would prevent our attack as well as any known microarchitectural side-channel attack. We repeat here the recommendation stated in RFC 7748 [200] as our attack uses a similar type of leakage from Libgcrypt's arithmetic library in order to achieve key extraction: "it is important that the arithmetic used not leak information about the integers modulo p, for example by having $b \cdot c$ be distinguishable from $c \cdot c$."

Rejecting Known Bad Points. To protect against small subgroup attacks against Curve-25519 and related curves that have a small set of low-order elements, an implementation can simply check if the received public key is in the set. Bernstein [56] provides a full list of these points for Curve25519, but suggests that rejecting these points is only necessary for protocols that wish to ensure "contributory" behavior. Langley and Hamburg [200] have a similar suggestion. We argue that rejecting these points would also give better side-channel protection. While this protection may seem unnecessary when used with constant-time code, as Kaufmann et al. [190] demonstrate, constant-time code is fragile and may fail to provide adequate protection.

Point Blinding. To protect the scalar k that is multiplied by a potentially-malicious ciphertext `P`, one can generate a random point `R`, compute $[k](\texttt{P} + \texttt{R})$, and then subtract $[k](\texttt{R})$ from the result [99]. This countermeasure completely protects against the chosen ciphertext attack we describe in this paper, since the attacker can no longer choose the point `P` to be multiplied with k. However, this countermeasure introduces an extra scalar-by-point multiplication for each decryption, so the negative performance effect of this countermeasure is significant.

Scalar Randomization. Many side-channel attacks rely on combining the leakage over

several decryption operations in order to extract the key. A possible countermeasure to prevent such averaging is scalar randomization [99], which adds a random multiple of the group order to the scalar k before performing the scalar-by-point multiplication operation. This changes the sequence of elliptic curve operations performed for every decryption operation, hindering the averaging operation. A similar countermeasure splits the scalar k into n parts $k_1, ...k_n$ such that $k = \sum_{i=1}^{n} k_i$, performs the scalar-by-point multiplication operation separately on each k_i, and then combines the result [94]. This countermeasure is cheaper than point blinding, but not as effective.

According to Bernstein [55], the order of the base point of Curve25519 is

$$2^{252} + 27742317777372353535851937790883648493.$$

We note that this number has a sequence of 128 consecutive zero bits. Ciet and Joye [94] note that scalar randomization with multipliers of this form still reveals a large number of bits. Thus, we do not recommend using this countermeasure.

Defense in Depth. The cache attack described in this paper will not work against an implementation that has truly constant-time code, since the attack relies on subtle timing differences deep within arithmetic functions. However, writing constant-time code is a non-trivial task; even the side-channel resistant Montgomery ladder algorithm still leaves room for error, as this paper demonstrates. Rather than providing the bare minimums for security, we argue that systems should be designed to have defense in depth, so that a single mistake on the part of the developer does not have disastrous consequences for security.

With regard to the attack described in this paper, the lack of input validation caused sensitive secret-key operations to be performed on adversarial inputs, which allowed us to transform an existing side-channel weakness into a full key-recovery attack. Thus, we recommend that in addition to writing side-channel resistant code, developers should also deploy the aforementioned countermeasures. This would have the effect of reducing the capability

of an attacker to mount key-extraction attacks by exploiting side-channel weaknesses.

4.6. Conclusion

In this work, we demonstrate a side-channel attack against Libgcrypt's implementation of ECDH encryption with Curve25519, which uses the Montgomery ladder and branchless formulas for point addition and doubling. Instead of relying on easily observable behavior such as high-level key-dependent branches or memory accesses, our attack exploits a low-level side channel vulnerability deep inside Libgcrypt's basic finite field arithmetic operations. We find that by passing order-4 elements into the decryption routine, we can trigger specific key-dependent code execution paths that a cache side channel attack is able to detect. From these key-dependencies, we are able to recover the key within about a second of measurements.

Chosen Ciphertext as Order-8 Element. While we did not investigate passing in order-8 elements as inputs to the decryption routine, these points would also introduce mathematical structure into the operands of the elliptic curve operations in the scalar-by-point multiplication. We expect that a similar attack would at least achieve partial key recovery.

Future Work. Our attack uses multiple decryption traces and averages the results to reduce the error rate. Overcoming side-channel noise to enable an attack with only a single trace is an open problem. Our attack relies on the special mathematical properties of the representation of the elements of order 4. Rejecting these points is an effective countermeasure to our attack; however, it does not address the underlying problem of having vulnerable arithmetic operations. It may be possible to extend our work to attack the arithmetic operations without using a low-order group element. Finally, our techniques should also be applicable for mounting low-bandwidth key extraction attacks against Libgcrypt's implementation of Curve25519 using physical side channels. Mounting such attacks remains an open problem.

CHAPTER 5 : 512-bit RSA in the wild

5.1. Introduction

A 512-bit RSA modulus was first factored by Cavallar et al. in 1999, which took about seven calendar months in a distributed computation using hundreds of computers and at least one supercomputer [88]. The current public factorization record, a 768-bit RSA modulus, was reported in 2009 by Kleinjung et al. [195]and took about 2.5 calendar years and a large academic effort.

Despite these successes, 512-bit RSA keys are still regularly found in use. Several implementations of the number field sieve have been published, including CADO-NFS [290], Msieve [251], and ggnfs [226], allowing even enthusiastic amateurs to factor 512-bit or larger RSA moduli. In 2009, Benjamin Moody factored a 512-bit RSA code signing key used on the TI-83+ graphing calculator using 2.5 calendar months of time on a single computer, and a distributed effort then factored several more 512-bit TI-68k and TI-Z80 calculator signing keys [281]. The NFS@Home project has organized several large distributed factorizations since 2009. [92] In 2012, Zachary Harris factored the 512-bit DKIM RSA keys used by Google and several other major companies in 72 hours per key using CADO-NFS and Amazon's Elastic Compute Cloud (EC2) service [314].

The persistence of 512-bit RSA is likely due in part to the legacy of United States policies regarding cryptography. In the 1990s, international versions of cryptographic software designed to comply with United States export control regulations shipped with 40-bit symmetric keys and 512-bit asymmetric keys, and export-grade cipher suites with these key sizes were built into protocols like SSL. Restrictions were later raised or lifted on open-source and mass-market software with cryptographic capabilities, but as of 2015, the United States Commerce Control List still includes systems "designed or modified to use 'cryptography' employing digital techniques performing any cryptographic function other than authentication, digital signature, or execution of copy-protected 'software' and having . . . an 'asym-

metric algorithm' where the security of the algorithm is based on ... factorization of integers in excess of 512 bits (e.g., RSA)". [85]

Factoring a 512-bit RSA key using the number field sieve is still perceived by many as a significant undertaking. In 2015, Beurdouche et al. [61] discovered the FREAK attack, a flaw in many TLS implementations that allows man-in-the-middle attacks to downgrade connections to 512-bit export-grade RSA cipher suites. In evaluating the prospect of a fully exploitable vulnerability, the paper states "we observe that 512-bit factorization is currently solvable at most in weeks." Subsequently, Bhargavan, Green, and Heninger developed a FREAK attack proof-of-concept in part by configuring CADO-NFS to run more efficiently on Amazon EC2. This setup was reported to factor a 512-bit key in approximately 7 hours on EC2, with a few additional hours for startup and shutdown [62].

In this paper, we present an improved implementation which is able to factor a 512-bit RSA key on Amazon EC2 in as little as four hours for $75. Our code is available at `https://github.com/eniac/faas`.

We gain these improvements by optimizing existing implementations for the case of factoring in the cloud. In particular, we rewrote the distributed portion of the number field sieve to use the Slurm job scheduler [312], allowing us to more effectively scale to greater amounts of computational resources. We describe our implementation and parallelizations in Section 5.3. We then performed extensive experiments on both CADO-NFS and Msieve to determine optimal parameter settings for the network interconnect speeds and resource limits achievable on Amazon EC2. Our experiments are detailed in Section 5.4.

Figure 6 summarizes the time and cost to factor a 512-bit RSA key using current optimal parameters with varying amounts of resources, and the average cost we paid between May and September 2015 for EC2 resources. By tuning the parameters for factoring, one can achieve different points in the trade-off between overall clock time and overall cost. Using more machines gives a faster overall factoring time, but has diminishing returns because

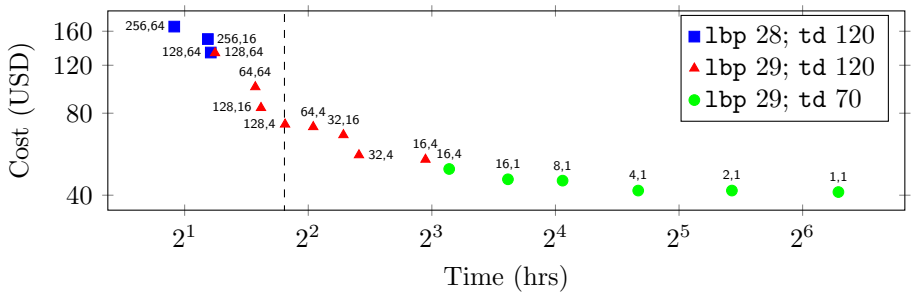


Figure 6: **A time/cost curve for 512-bit factorization**—Each point above is annotated with the instances used for sieving and linear algebra, respectively, and represents an experimental estimate. There are diminishing returns from imperfect parallelization in linear algebra. The dotted line shows the fastest time we were able to achieve; larger experiments usually encountered node instability.

of imperfect parallelism. Linear algebra time was measured empirically and sieving was measured once for each parameter set and extrapolated to different numbers of instances.

The order of magnitude of the costs we give lines up with previous reports and estimates of factoring on EC2, and we achieve a significant speedup in overall running time. Performing a computation of this magnitude reliably remains a challenging endeavor. Our paper can also be viewed as a case study on the successes and challenges in trying to replicate a high-performance computing environment in the Amazon EC2 cloud.

In order to measure the impact of fast 512-bit factorization, in Section 5.5 we analyze existing datasets and perform our own surveys to quantify 512-bit RSA key usage in modern cryptographic public key infrastructures. We find thousands of DNSSEC records signed with 512-bit keys, millions of HTTPS, SMTP, IMAPS, and POP3S servers still supporting `RSA_EXPORT` cipher suites for TLS, and a long tail of 768-bit, 512-bit, and shorter RSA keys in use across DKIM, SSH, IPsec VPNs, and PGP.

5.2. Background

In this paper, we focus on the impact of factoring on the security of RSA public keys [270], though integer factorization has many applications across mathematics. Factoring the modulus of an RSA public key allows an attacker to compute the corresponding private key, and thus to decrypt any messages encrypted to that key, or forge cryptographic signatures

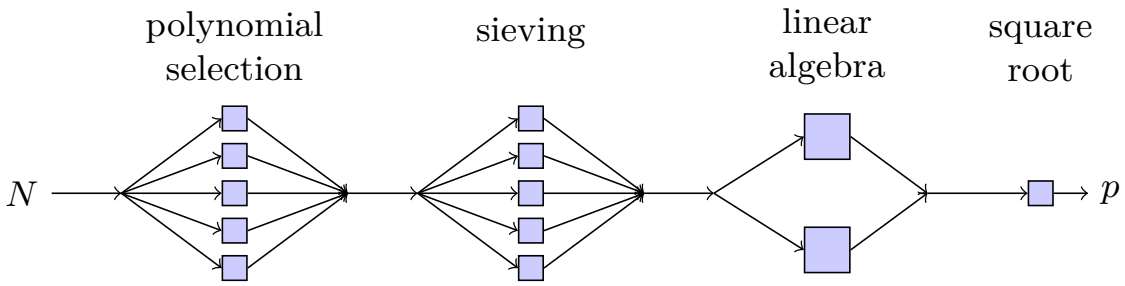


Figure 7: **The number field sieve**—The number field sieve factoring algorithm consists of several main stages. Sieving and linear algebra are the most computationally intensive stages. Sieving is embarrassingly parallel, while parallelizing linear algebra can encounter communication bottlenecks.

using the private key.

5.2.1. Number Field Sieve

The general number field sieve is the fastest known algorithm for factoring generic integers larger than a few hundred bits [204]. Its running time is described using L-notation as

$$L_N[1/3, 1.923] = \exp\left(1.923(\log N)^{1/3}(\log\log N)^{2/3}\right),$$

which is sub-exponential, but super-polynomial [171] in the size of N, the integer to be factored. A gentle introduction to the big ideas behind sieving algorithms for integer factorization and can be found in Pomerance's 1996 survey [264], and more in-depth information on the number field sieve can be found in the books by Lenstra, Lenstra, Manasse, and Pollard [204] and Crandall and Pomerance [101].

In this section, we give a brief overview of the structure of the algorithm, in order to identify potential implementation optimizations and barriers to parallelization. The number field sieve has four main computational stages: *polynomial selection*, *sieving*, *linear algebra*, and *square root*.

The first stage of the algorithm, *polynomial selection*, searches for a polynomial $f(x)$ and integer m satisfying $f(m) \equiv 0 \bmod N$, where N is the integer to factor. $f(x)$ defines the number field $\mathbb{Q}(x)/f(x)$ to be used in the rest of the algorithm. A good choice of polynomial

in this stage can significantly speed up the rest of the computation, by generating smaller elements in the sieving phase. Several techniques exist for choosing the polynomial, but in general many different polynomials are tested and the best one is passed on to the next stage. The polynomial selection stage is embarrassingly parallel.

The next stage of the algorithm, *sieving*, factors ranges of integers and number field elements to find many relations of elements and saves those whose prime factors have size less than some size bound B, called the smoothness bound. CADO-NFS uses the *large prime variant* of sieving, and the large prime bound parameters `lbp` control the log of the smoothness bounds. Decreasing these bounds increases the difficulty of sieving, since relations are less likely to factor completely into smaller factors. The sieving stage is also embarrassingly parallel, since candidate relations can be evaluated independently in small batches.

In the third stage, *linear algebra*, the coefficient vectors of the relations are used to construct a large sparse matrix with entries over $\mathbb{F}_2$. Before beginning this stage, some preprocessing on the relations is used to decrease the dimension of the resulting matrix. In general, more relations collected during sieving will produce a smaller matrix and reduce the runtime for linear algebra. The goal of the linear algebra stage is to discover a linear dependency among the rows. This is accomplished via the Block Wiedemann [98] or Block Lanczos [228] algorithms, which are specialized for sparse linear algebra. This step can be parallelized, but the parallelization requires much more communication and synchronization.

The final stage involves computing the *square root* of a number field element corresponding to a dependency in the matrix. In practice, many dependencies will be tested since not all of them will lead to a nontrivial factor; the square roots can be computed and tested in parallel. This step takes only a few minutes.

Discrete log. There is also a number field sieve algorithm for discrete logarithms with a nearly identical structure. Many of the implementation improvements that we describe here also apply to discrete log. However a 512-bit prime-field discrete log is significantly

more burdensome than a 512-bit factorization, in large part because the linear algebra stage involves arithmetic over a large-characteristic finite field. Adrian et al. [29] describe 512-bit discrete log computations in practice; we estimate that a single equivalent discrete log computation performed on Amazon EC2 would cost approximately $1400 and take 132 hours.

5.2.2. Amazon EC2

Amazon Elastic Compute Cloud (EC2) is a service that provides virtualized computing resources that can be rented by the hour. Several competitors exist, including Google Compute Engine. We specialize our results to Amazon largely out of convenience and because when we began this project some tools were specialized to Amazon's infrastructure.

Amazon EC2 bills for computing resources by the *instance-hour*. An *instance* is a single virtualized machine associated with resources including processing cores, memory, and disk storage. Amazon offers many different instance types. We chose the largest type of compute-optimized instance available as of August 2015, the `c4.8xlarge` instance. This instance type has two Intel Xeon E5-2666 v3 processor chips, with 36 vCPUs in a NUMA configuration with 60 GB of RAM.

There are multiple pricing structures available to purchase instance-hours. For our purposes, one can purchase fixed-rate *on-demand* instances, or bid a variable rate for *spot instances* which may be terminated depending on demand. The difference can be significant: for a `c4.8xlarge` instance, the on-demand price as of September 2015 is $1.763, while the average spot price we paid between May and September 2015 was $0.52. We used spot instances for our experiments. Amazon raised our account limit to allow us to launch up to 200 instances.

The `c4.8xlarge` instance type supports Enhanced Networking with 10 GbE interconnect between instances. Machines can be rented in different *availability zones* located around the world, and within an availability zone one can request machines to be co-located in a single *placement group* to minimize latency. We measured the interconnect bandwidth of

instances in the same availability zone and placement group at 9.46 Gbit/s, and between instances not in the same placement group at 4–5 Gbit/s. We enabled enhanced networking and launched instances used for linear algebra in one placement group.

The networking environment of Amazon EC2 is distinct from a traditional HPC cluster. The connection was not saturated during our linear algebra optimization tests in Section 5.4 below. However, our measured interconnect latency, at 151 μs, is significantly greater than most HPC standards. For reference, InfiniBand FDR has latency requirements of 7 μs at 10 Gbit speeds.

Kleinjung, Lenstra, Page, and Smart [196] estimated in 2012 that factoring 512-bit RSA on Amazon EC2 would cost $107 for sieving and $30 for linear algebra. Their estimates were obtained from experiments on truncated sieving jobs and simplified linear algebra. In comparison, we focused on building a system to reliably perform full 512-bit factorizations as quickly as possible given the current state of the EC2 platform. Paterson, Poettering, and Schuldt [252] used EC2 to perform large-scale cryptanalytic experiments for the RC4 stream cipher.

5.3. Implementation

In order to speed up factoring, we wanted to maximize parallelism. In the polynomial selection and sieving stages, parallelization is straightforward, because the tasks can be split into arbitrarily small pieces to be executed independently, with only a relatively small amount of sequential work to process the results together at the end. Our improvements in these stages come from reliably distributing these tasks across cluster resources in a scalable way. Scaling the linear algebra stage is more complex, because the communication overhead results in diminishing returns from additional resources. We performed extensive experiments to characterize the trade-offs and guide parameter selection.

5.3.1. Managing Amazon EC2 resources with Ansible

We used Ansible [104], a cluster management tool, to set up and configure an EC2 cluster and to scale the cluster appropriately at each stage of factorization. After the sieving

stage, we terminate nodes not required for linear algebra. Ansible can launch and configure a cluster of 50 on-demand instances in under 5 minutes, and 50 spot instances in 10–15 minutes.

5.3.2. Parallelizing polynomial selection and sieving with Slurm

The polynomial selection and sieving stages generate thousands of individual tasks to be distributed to cluster compute nodes. This requires a job distribution framework that is fast and scalable to many machines. The CADO-NFS implementation is distributed with a Python script to coordinate each stage, including a job distribution system over HTTP designed to require minimal setup from participating computers. Unfortunately this implementation did not scale well to simultaneously tracking thousands of tasks. We experimented with Apache Spark [313] to manage data flow, but Spark was not flexible enough for our needs, and our initial tests suggested that a Spark-based job distribution system was more than twice as slow as the system we were aiming to replace.

Ultimately we chose Slurm (Simple Linux Utility for Resource Management) [312] for job distribution and management during polynomial selection and sieving. Slurm can resubmit failed or timed-out tasks, monitors for and deals with failed nodes, has low startup overhead, and scales well to large clusters.

Our implementation uses a management thread to submit polynomial selection and sieving tasks asynchronously in batches to the Slurm controller, which then handles distribution and execution. This thread rate limits batch sizes in order to get around Slurm's job submission rate of a thousand jobs per second. [41] We found that scheduling two jobs per vCPU yielded faster sieving times than one job per vCPU, since the latter did not always fully saturate CPU usage.

5.3.3. Parallelizing linear algebra with MPI

After sieving has completed, the relations that have been produced are processed to generate a large, sparse matrix. The runtime of this linear algebra phase depends on the dimension of the matrix and the number of nonzero entries per matrix row, called the *density*, so the

preprocessing stage attempts to produce a matrix that is as small as possible by filtering and combining relations. The parameters that control the effectiveness of the dimension reduction are the number of relations collected and the allowed density of the matrix.

The parallelization of the linear algebra stage is more complex than sieving or polynomial selection. In general, the matrix is divided up into an $n \times n$ grid. In each iteration, each worker operates on its own grid element, gathers results from each of the other workers using the Message Passing Interface (MPI), and combines the results into its own grid element. We used OpenMPI 1.8.6. [133]

Comparing CADO-NFS and Msieve linear algebra. We compared the linear algebra implementations of CADO-NFS, which implements the Block Wiedemann algorithm, and Msieve, which implements the Block Lanczos algorithm for linear algebra. Although Block Wiedemann is designed to parallelize well on independent resources, Msieve was significantly faster on our EC2 configuration. Both implementations support MPI out of the box. For a 512-bit factorization with an identical set of 53 million relations, we found that CADO-NFS without MPI completed the linear algebra stage in 350 minutes, while Msieve without MPI completed linear algebra in 140 minutes. When parallelized across multiple EC2 instances, CADO-NFS's runtime did not decrease significantly, whereas Msieve's did. We decided to use Msieve's implementation for linear algebra.

Unfortunately, the input and output formats used by CADO-NFS and Msieve are not compatible, so using Msieve's linear algebra meant we also needed to use Msieve's matrix preprocessing and final square root phases or rewrite these stages ourselves. We compromised by parallelizing Msieve's square root implementation to test multiple dependencies simultaneously, so that the square root phase finishes in approximately 10 minutes.

5.4. Experiments

We performed several experiments to explore the effects of different parameter settings on running time. All of the experiments in this section were carried out on the same arbitrarily chosen 512-bit RSA modulus. There will be some variation in running time across different

lbp	relations	matrix rows	matrix size	sieve CPU-hours	linalg instance-hours
28	28.2M	4.96M	1.48 GB	3271.1	5.4
29	44.8M	5.68M	1.71 GB	2369.2	8.5

Table 25: **Large prime bounds**—Decreasing the large prime bound parameter increases the amount of work required for sieving, but decreases the work required for linear algebra. This is an advantageous choice when large amounts of resources can be devoted to sieving.

moduli. In order to understand this variation, we measured the CPU time required to sieve 54.5 million relations for five different randomly generated RSA moduli with the parameters `lbp` 29 and target density 70 on a cluster with 432 CPUs. We observed a median of 2770 CPU hours with a standard deviation of 227 CPU hours in the sample set.

5.4.1. Large prime bounds

The large prime bounds `lbp` specify the log of the smoothness bound for relations collected in the sieving stage. Decreasing the large prime bound will decrease the dimension of the matrix and therefore decrease the linear algebra running time, but will increase sieving time because relations with smaller prime factors are less common. The `lbp` parameter provides the first step for tuning the trade-off between sieving and linear algebra time to optimize for different-sized clusters.

We experimented with `lbp` values 28 and 29. At `lbp` 27, CADO-NFS was unable to gather enough relations even after increasing the sieving area. At `lbp` 30, linear algebra will dominate the computation time even for small clusters.

Table 25 shows the effect of the changing the large prime bound for one experimental setup. Both of the runs used the minimum number of relations required to build a full matrix with target density 70 (see Section 5.4.2), and linear algebra was completed on a single machine with 36 vCPUs. Decreasing `lbp` from 29 to 28 causes the sieving CPU time to increase by 38% even though fewer relations are collected, but the linear algebra time decreases by 36%.

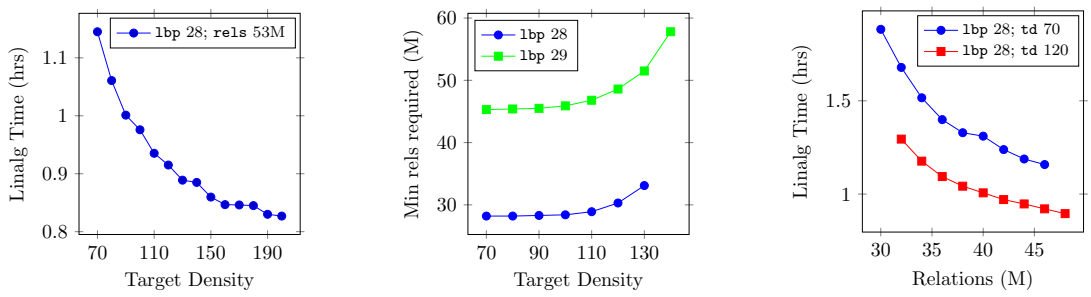


Figure 8: **Target density and oversieving**—Increasing the target density parameter decreases linear algebra time, but requires more relations to construct the matrix. Collecting additional relations beyond the minimum also produces a better matrix and decreases linear algebra time. This trade-off can be advantageous if more resources can be devoted to sieving, as sieving parallelizes well.

5.4.2. Target density

The target density parameter specifies the average number of sparse nonzero entries per matrix row that Msieve will aim for in matrix construction. Linear algebra time is dependent on the product of the density and dimension, and can be decreased by raising the target density to lower the dimension. Figure 8a shows how increasing the target density decreases linear algebra time for a fixed set of input relations on a cluster of 16 instances.

For a 512 bit number with 53 million relations (more than 20 million relations over the minimum), a matrix with target density 70 took 15 minutes to construct and 68 minutes for the linear algebra computation. For the same set of relations, a matrix with target density of 120 took 17 minutes to construct and 55 minutes for linear algebra, a 19% reduction in linear algebra time. However, there were diminishing returns to increases in target density: increasing the target density from 120 to 170 reduced the overall time by only 4%.

The drawback to increasing target density is that more relations are needed from the sieving stage to construct the matrix. Figure 8b shows how the minimum number of relations required increases sharply as target density is increased beyond a particular threshold. When large amounts of resources are available for sieving, the increased work required to

collect additional relations can be compensated for by a larger decrease in linear algebra time. For a given cluster size, there is an optimal target density that takes into account these trade-offs.

5.4.3. Oversieving

Oversieving means generating excess relations during the sieving phase. This can help to produce an easier matrix for the linear algebra phase, reducing linear algebra runtime. We ran experiments varying cluster configurations, target densities, and large prime bounds to determine an oversieving curve for each. Figure 8c shows two representative oversieving curves for a 16-node linear algebra cluster with `lbp` 28 and target densities 70 and 120, respectively. For the target density 70 curve, the linear algebra time for the minimum number of relations required to construct the matrix, 30 million, was 112 minutes. At 32 million relations, the linear algebra time was reduced to 101 minutes, an 11% improvement. However, as Figure 8c shows, there are diminishing returns to oversieving, while the work required to produce additional relations scales close to linearly. Optimal oversieving amounts are dependent on the cluster configuration.

5.4.4. MPI grid size

The grid size parameter directly controls the number of work units that MPI can assign to cluster resources. We experimented with both fine-grained grids matching the number of work units to the total number of vCPUs, and coarse-grained grids matching work units to instances. The optimum turned out to be somewhere in the middle: a single multithreaded work unit was not able to occupy all of the 36 vCPUs on a single instance, while the other extreme is likely to become limited by communication overhead since the Block Lanczos algorithm requires each node to gather results from every other node at each iteration.

In order to determine the optimal grid size, we tested a range of grid sizes for cluster sizes of 1, 4, 16, and 64 instances. The best performance for clusters with 1 and 4 instances was 4x4 and 8x8, respectively, where each cluster had 16 work units in total. For the clusters with 16 and 64 instances, the optimal grid size was 8x8 and 16x16, where each cluster had 4 work units in total. The differences as cluster size grows are likely due to communication

bottlenecks.

5.4.5. Processor affinity

The default parameters of OpenMPI dictate that each of the work units is bound to a specific machine, but when multiple work units are assigned to the same instance they compete for the same processor and memory resources, creating processor scheduling overhead and increased variance in the work unit iteration times. Each work unit must iterate together, so the time per iteration is dictated by the slowest work unit. Since the `c4.8xlarge` EC2 instances have two processor sockets and a NUMA memory layout, the distribution of the threads of a work unit across two processors means longer intra-process communication times and slower memory access times. We used the rankfile/process affinity parameter in OpenMPI to bind each of the work units on a single instance to its own subset of processor cores and saw an improvement of 1-2% in linear algebra time.

We also tested binding each thread of each of the work units to individual cores, but this did not improve running times.

5.4.6. Block size

The default block size in Msieve is 8192 bytes. Theoretically, matching the block size used in Msieve with the size of the L1 cache of the processor should yield better performance by decreasing cache and memory access times. However, for the parameters `lbp` 28 and target density 70, increasing the block size from 8K to 16K increased computation time from 67 minutes to 69 minutes, and increasing the block size from 8K to 32K increased computation time from 67 minutes to 73 minutes. We decided to leave the block size unchanged.

5.4.7. Putting it all together

To generate the data points in Figure 6, we individually timed each sieving job together with system overhead. For each set of parameters, we combined the linear algebra running time from the experiments in this section with the total measured running time to complete enough sieving jobs to generate the required number of relations. We then added a measured estimate of costs for the remaining steps of factoring to get our total running time estimates. We were able to reliably achieve running times under four hours for factoring, but in several

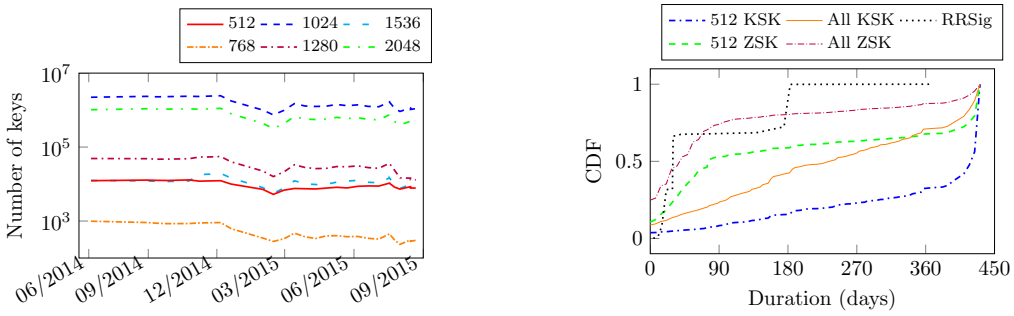
attempts to verify lower overall times, we encountered issues where some EC2 instances in our cluster ran more slowly than others or became unresponsive. These issues become more pronounced with larger cluster sizes. Our sieving setup can deal gracefully with slow nodes, but linear algebra is more fragile and is currently limited by the slowest node.

5.5. 512-bit keys still in use

In this section, we survey RSA key lengths across public key infrastructures for a variety of protocols, finding that 512-bit RSA keys are surprisingly persistent.

5.5.1. DNSSEC

DNSSEC [40] is a DNS protocol extension that allows clients to cryptographically authenticate DNS records. DNS records protected by DNSSEC include a public key record (usually RSA) and a signature that can be chained up to a trusted root key. DNSKEY records can contain either a zone-signing key (ZSK), used to sign DNS records, or a key-signing key (KSK), used to sign DNSKEY records. RFC 4033 [40] specifies that zone-signing keys may have shorter validity periods, and key-signing keys should have longer validity periods. RFC 6781 [199], published by the IETF in 2012 on DNSSEC Operational Practices, states that "it is estimated that most zones can safely use 1024-bit keys for at least the next ten years."



(a) Key sizes

(b) Key and signature durations

Figure 9: **DNSSEC key sizes and duration**—The ratios of RSA key lengths has remained relatively stable over time, although the total number of DNSSEC keys collected fluctuated across scans. The number of 512-bit keys remained around 10,000, or 0.35% of the total. Many DNSSEC keys are rotated infrequently, and 512-bit keys are rotated less frequently than longer keys.

An attacker who knows the private key to a zone-signing key or key-signing key could mount an active attack to forge DNS responses for any descendants below that location in the chain.

We analyzed several DNSSEC datasets. The most comprehensive is a collection of DNS records collected by Rapid7 which we downloaded from Scans.io. They performed biweekly DNS lookups on approximately 529 million domains starting in June 2014 and continuing to present. The number of lookups varies by as much as 61 million domains across scans, and the number of domains with valid DNSSEC records fluctuated between 3.7 million and 1.1 million and decreased over time compared to total domains. The relative fraction of DNSSEC key sizes did not change much over time. The distribution is shown in Figure 9a.

In order to measure the completeness of the Rapid7 dataset, we compared to a second dataset of anonymized 512-bit DNSSEC keys for all .com, .net, and .org domains between February 22, 2015 and September 3, 2015 from the SURFnet DNS measurement infrastructure of van Rijswijk-Deij, Jonker, Sperotto, and Pras [301] which was provided to us by the researchers. The SURFnet data contained 2,116 distinct public keys of which 1,839 (86%) were present in the Rapid7 scans from the same time period. To measure how many 512-bit keys are in active use, SURFnet provided a set of all 512-bit DNSkey records collected using their passive DNS monitoring system for a one-month period between September 12, 2015 and October 13, 2015. The set included 1,239 records covering 613 distinct domains and contained 705 distinct keys.

Finally, we performed DNS lookups on eleven thousand zones not contained in the Rapid7 dataset that were required for signature validation. 56% of domains with 512-bit keys failed signature verification, most commonly because the TLD signature was not present in the chain of trust.

Many keys were never rotated at all over the 431-day period spanned by the Rapid7 dataset, and signatures were renewed more frequently than keys were updated. Figure 9b illustrates

Length	All Certificates	Distinct Keys	Trusted Certificates	Trusted and Valid
512	303,199 (0.9%)	32,870	0 (0.0%)	0 (0.0%)
768	26,582 (0.1%)	14,581	0 (0.0%)	0 (0.0%)
1024	12,541,661 (36.8%)	3,196,169	4,016 (0.0%)	4,012 (0.0%)
1536	2,537 (0.0%)	2,108	0 (0.0%)	0 (0.0%)
2048	20,782,686 (60.9%)	6,891,678	14,413,589 (42.2%)	14,411,618 (42.2%)
2432	2,685 (0.0%)	1,191	128 (0.0%)	128 (0.0%)
3072	65,765 (0.2%)	58,432	1,787 (0.0%)	1,787 (0.0%)
4096	391,123 (1.1%)	218,334	259,898 (0.8%)	259,830 (0.8%)
8192	2,172 (0.0%)	971	481 (0.0%)	481 (0.0%)
RSA Export	2,630,789 (7.7%)			
Total	34,121,474 (100.0%)		14,680,782 (43.0%)	14,678,739 (43.0%)

Table 26: **HTTPS RSA common key lengths and export RSA support**—HTTPS scans downloaded from `scans.io` were performed using ZMap on port 443 on August 23 and September 1, 2015.

signature validity periods and key lifetimes. Signature validity periods are clustered around a few common ranges: 33% of keys were signed for six months, 34% percent for one month, 25% for three weeks, and 6% for 14 days. 512-bit zone-signing keys and key-signing keys were less frequently rotated than other key sizes.

5.5.2. HTTPS

RSA public keys are used for both encryption and authentication in the TLS protocol. If the client and server negotiate an RSA cipher suite, the client encrypts the premaster secret used to derive the session keys to the RSA public key in the server's certificate. An adversary who compromises the private key can passively decrypt session traffic from the past or future. However, since no 512-bit certificates have currently valid signatures from certificate authorities, these servers are also vulnerable to an active man-in-the-middle attack from an adversary who simply replaces the certificate.

If the client and server negotiate a Diffie-Hellman or elliptic curve Diffie-Hellman cipher suite, the server uses the public key in its certificate to sign its key exchange parameters. An adversary who knows the private key could carry out a man-in-the-middle attack by forging a correct signature on their desired parameters. Since again no 512-bit certificates are currently signed or trusted, such an active adversary could also merely replace the server

	Port	Handshake	RSA_EXPORT	512-bit Certificate Key
SMTP	25	4,821,615	1,483,955 (30.8%)	64 (0%)
IMAPS	993	4,468,577	561,201 (12.6%)	102 (0%)
POP3S	995	4,281,494	558,012 (13.0%)	115 (0%)

Table 27: **Mail protocol key lengths**—An Internet-wide scan of TLS usage in three common mail protocols shows higher levels of support for RSA_EXPORT cipher suites than in HTTPS.

certificate in the exchange along with the chosen Diffie-Hellman parameters.

Finally, connections to servers supporting `RSA_EXPORT` cipher suites may be vulnerable to an active downgrade attack if the clients have not been patched against the FREAK attack. [61] Successfully carrying out this attack requires the attacker to factor the server's ephemeral RSA key, which is typically generated when the server application launches and is reused as long as the server is up. "Ephemeral" RSA keys can persist for weeks and are almost always 512 bits.

We examined IPv4 scan results for HTTPS on port 443 performed using ZMap [114] by the University of Michigan which we accessed via Scans.io and the Censys scan data search interface developed by Durumeric et al. [117]. Table 26 summarizes scans from August 23 and September 1, 2015.

Durumeric, Kasten, Bailey, and Halderman [113] examined the HTTPS certificate infrastructure in 2013 using full IPv4 surveys and found 2,631 browser-trusted certificates with key lengths of 512 bits or smaller, of which 16 were valid. Heninger, Durumeric, Wustrow, and Halderman [166] performed a full IPv4 scan of HTTPS in October 2011 with responses from 12.8 million hosts, and found 123,038 certificates (trusted and non-trusted) containing 512-bit RSA keys. Similar to [166], we observe many repeated public keys.

5.5.3. Mail

Table 27 summarizes several Internet-wide scans targeting SMTP, IMAPS, and POP3S. The scans were performed by the University of Michigan using ZMap between August 23, 2015, and September 3, 2015.

Length	Keys
4096	5 (0.0%)
2048	64 (0.5%)
1028	1 (0.0%)
1024	10,726 (92.2%)
768	126 (1.1%)
512	103 (0.9%)
384	20 (0.2%)
128	1 (0.0%)
Parse error	591 (5.1%)
Total	11,637

Table 28: **DKIM key sizes**—DKIM public keys were collected from a Rapid7 DNS dataset, and manual DNS lookups of 11,600 domains containing DKIM records that we performed on September 4, 2015.

We used the Censys scan database interface provided by [117] to analyze the data. While only a few hundred few mail servers served TLS certificates containing 512-bit RSA public keys, 13% of IMAPS and POP3S servers and 30% of SMTP servers supported `RSA_EXPORT` cipher suites with 512-bit ephemeral RSA, meaning that unpatched clients are vulnerable to the FREAK downgrade attack by an adversary with the ability to quickly factor a 512-bit RSA key.

We also examined DKIM public keys. DomainKeys Identified Mail [35] is a public key infrastructure intended to prevent email spoofing. Mail providers attach digital signatures to outgoing mail, which recipients can verify using public keys published in a DNS text record.

We gathered DKIM public keys from the Rapid7 DNS dataset. However, the published dataset had lowercased the base64-encoded key entries, so we performed DNS lookups on the 11,600 domains containing DKIM records ourselves on September 4, 2015. We made a best-effort attempt to parse the records, but 5% of the responses contained a key that was malformed or truncated and could not be parsed. Of the remainder, 124 domains used 512-bit keys or smaller, including one that used a 128-bit RSA public key. We were able to

Length	Keys
4096	37 (0.8%)
3072	1 (0.0%)
2048	2,257 (51.3%)
1024	1,804 (41.0%)
768	1 (0.0%)
512	69 (1.6%)
Parse error	234 (5.3%)
Total	4,403 (100%)

Table 29: **IPsec VPN certificate keys**—We performed Internet-wide scans on port 500 of IKEv1 in aggressive mode. Of the servers that responded with certificates, 1.6% had a 512-bit public key.

factor this key in less than a second on a laptop and verify that it is, in fact, a very short RSA public key. Table 28 summarizes the distribution.

Durumeric et al. [118] surveyed cryptographic failures in email protocols using Internet-wide scans and data from Google. They examine DKIM use in April 2015 and discovered that 83% of mail received by Gmail contained a DKIM signature, but of these, 6% failed to validate. Of these failures, 15% were due to a key size of less than 1024 bits, and 63% were due to other errors.

5.5.4. IPsec

We conducted two ZMap scans of the full IPv4 space to survey key sizes in use by IPsec VPN implementations that use RSA signatures for identity validation during server-client handshakes. An adversary who compromised the private keys for one of these certificates could mount a man-in-the-middle attack.

Our ZMap scans targeted IKEv1 aggressive mode [164], which allows the server to send a certificate after a only a single message is received. The messages we sent contained proposals for DES, 3DES, AES-128, and AES-256 each with both SHA1 and MD5. Our first scan offered a key exchange using Oakley group 2 (a 1024-bit Diffie-Hellman group) and elicited certificates from 4% of the servers that accepted our message. Our second scan

RSA Size	Hosts	Distinct
512	508 (0.0%)	316
768	2,972 (0.0%)	2,419
784	3,119 (0.0%)	223
1020	774 (0.0%)	572
1024	296,229 (4.4%)	91,788
1040	2,786,574 (41.3%)	1,407,922
1536	639 (0.0%)	536
2048	3,632,865 (53.9%)	1,752,406
2064	1,612 (0.0%)	957
4096	15,235 (0.2%)	1,269
RSA Total	6,741,352	3,258,742
DSA	692,011	421,944
ECDSA	2,192	2,192

Table 30: **SSH host key lengths**—Host keys were collected in April 2015 in a ZMap scan of SSH hosts on port 22 mimicking OpenSSH 6.6.1p1.

offered Oakley group 1 (a 768-bit Diffie-Hellman group) and received responses from 0.2% of hosts. Of the non-responses from both scans, 71% of the servers responded indicating that they did not support our combination of aggressive mode with our chosen parameters, 16% rejected our connection for being unauthorized (not on a whitelist), and the remaining 11% returned other errors. As shown in Table 29, 1.6% of certificates collected had a 512-bit public key.

5.5.5. SSH

SSH hosts authenticate themselves to the client by signing the protocol handshake with their public host key. Clients match the host key to a stored trusted fingerprint. An adversary who is able to compromise the private key for an SSH host key can perform an active man-in-the-middle attack.

Table 30 summarizes host key sizes collected by a ZMap scan of SSH hosts on port 22 mimicking OpenSSH 6.6.1p1. The data was collected in April 2015 by Adrian et al. [29], who provided it to us. A very large number of hosts used 1040-bit keys; these hosts had banners identifying them as using Dropbear, a lightweight SSH implementation aimed at embedded

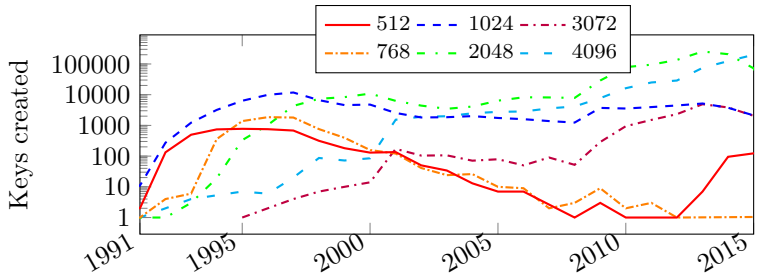


Figure 10: **PGP RSA public key lengths by reported creation date**—RSA public keys were downloaded from `keyserver.borgnet.us`, a PGP keyserver bootstrap dataset, on October 4, 2015.

devices. Heninger et al. [166] performed a full IPv4 scan of SSH public keys in February 2012 offering only `Diffie-Hellman Group 1` key exchange. Of 10 million responses, they reported that 8,459 used 512-bit RSA host keys and observed many repeated host keys.

Clients can also use public keys to authenticate themselves to a server. An adversary who is able to compromise the private key for a client SSH authentication key can access the server by logging in as the client. Cox [100] collected 1,376,262 SSH public keys that had been uploaded to GitHub by users to authenticate themselves to the service between December 2014 and January 2015 by using GitHub's public API. He collected 1,205,330 RSA public keys, 27,683 DSA public keys, and 1,060 ECDSA public keys. Of the RSA public keys, 2 had 256-bit length, 3 had 512-bit length, and 28 had 768-bit length.

5.5.6. PGP

PGP implements encryption and digital signatures on email or files. RSA public keys can be used for both encryption and signatures. PGP uses a public "web of trust" model: users can distribute their public keys along with signatures attesting trust relationships via a public network of keyservers. An adversary who compromises a PGP public key could use it to impersonate a user with a digital signature or decrypt content encrypted to that user.

We downloaded a PGP keyserver bootstrap dataset from `keyserver.borgnet.us` on October 4, 2015. It contained 4.9 million public keys from 3 million users. Of these, 1.6 million were RSA, 1.7 million were DSA, 1.7 million were ElGamal, 398 were ECDH, 158 were

EdDSA, and 513 were ECDSA. 4,688 512-bit RSA keys were present in the dataset; 123 of them listed a creation date in 2015. Figure 10 shows the shift to longer RSA key lengths over time.

5.6. Conclusions

512-bit RSA has been known to be insecure for at least fifteen years, but common knowledge of precisely *how* insecure has perhaps not kept pace with modern technology. We build a system capable of factoring a 512-bit RSA key in under four hours. We then measure the impact of such a system by surveying the incidence of 512-bit RSA in modern cryptographic infrastructure, and find a long tail of too-short public keys and export-grade cipher suites still in use in the wild. These numbers illustrate the challenges of keeping an aging Internet infrastructure up to date with even decades-old advances in cryptanalysis.

CHAPTER 6 : Logjam attack and measurements

6.1. Introduction

Diffie-Hellman key exchange is widely used to establish session keys in Internet protocols. It is the main key exchange mechanism in SSH and IPsec and a popular option in TLS. We examine how Diffie-Hellman is commonly implemented and deployed with these protocols and find that, in practice, it frequently offers less security than widely believed.

There are two reasons for this. First, a surprising number of servers use weak Diffie-Hellman parameters or maintain support for obsolete 1990s-era export-grade crypto. More critically, the common practice of using standardized, hard-coded, or widely shared Diffie-Hellman parameters has the effect of dramatically reducing the cost of large-scale attacks, bringing some within range of feasibility today.

The current best technique for attacking Diffie-Hellman relies on compromising one of the private exponents (a, b) by computing the discrete log of the corresponding public value ($g^a \bmod p$, $g^b \bmod p$). With state-of-the-art number field sieve algorithms, computing a single discrete log is more difficult than factoring an RSA modulus of the same size. However, an adversary who performs a large precomputation for a prime p can then quickly calculate arbitrary discrete logs in that group, amortizing the cost over all targets that share this parameter. Although this fact is well known among mathematical cryptographers, it seems to have been lost among practitioners deploying cryptosystems. We exploit it to obtain the following results:

Active attacks on export ciphers in TLS. We introduce Logjam, a new attack on TLS by which a man-in-the-middle attacker can downgrade a connection to export-grade cryptography. This attack is reminiscent of the FREAK attack [61] but applies to the ephemeral Diffie-Hellman ciphersuites and is a TLS protocol flaw rather than an implementation vulnerability. We present measurements that show that this attack applies to 8.4% of Alexa Top Million HTTPS sites and 3.4% of all HTTPS servers that have browser-trusted certifi-

cates.

To exploit this attack, we implemented the number field sieve discrete log algorithm and carried out precomputation for two 512-bit Diffie-Hellman groups used by more than 92% of the vulnerable servers. This allows us to compute individual discrete logs in about a minute. Using our discrete log oracle, we can compromise connections to over 7% of Top Million HTTPS sites. Discrete logs over larger groups have been computed before [75], but, as far as we are aware, this is the first time they have been exploited to expose concrete vulnerabilities in real-world systems.

We were also able to compromise Diffie-Hellman for many other servers because of design and implementation flaws and configuration mistakes. These include use of composite-order subgroups in combination with short exponents, which is vulnerable to a known attack of van Oorschot and Wiener [300], and the inability of clients to properly validate Diffie-Hellman parameters without knowing the subgroup order, which TLS has no provision to communicate. We implement these attacks too and discover several vulnerable implementations.

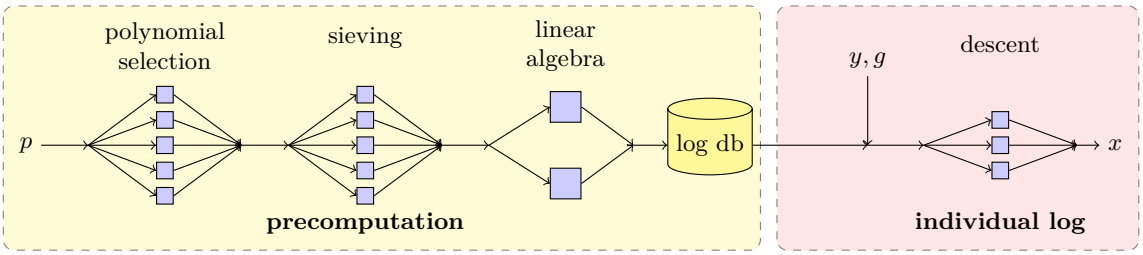


Figure 11: **The number field sieve algorithm for discrete log**—The algorithm consists of a precomputation stage that depends only on the prime p and a descent stage that computes individual logs. With sufficient precomputation, an attacker can quickly break any Diffie-Hellman instances that use a particular p.

Risks from common 1024-bit groups. We explore the implications of precomputation attacks for 768- and 1024-bit groups, which are widely used in practice and still considered secure. We provide new estimates for the computational resources necessary to compute discrete logs in groups of these sizes, concluding that 768-bit groups are within range of

academic teams, and 1024-bit groups may plausibly be within range of state-level attackers. In both cases, individual logs can be quickly computed after the initial precomputation.

We then examine evidence from published Snowden documents that suggests NSA may already be exploiting 1024-bit Diffie-Hellman to decrypt VPN traffic. We perform measurements to understand the implications of such an attack for popular protocols, finding that an attacker who could perform precomputations for ten 1024-bit groups could passively decrypt traffic to about 66% of IKE VPNs, 26% of SSH servers, 16% of SMTP servers, and 24% of popular HTTPS sites.

Mitigations and lessons. As a short-term countermeasure in response to the Logjam attack, all mainstream browsers are implementing a more restrictive policy on the size of Diffie-Hellman groups they accept. We further recommend that TLS servers disable export-grade cryptography and carefully vet the Diffie-Hellman groups they use. In the longer term, we advocate that protocols migrate to stronger Diffie-Hellman groups, such as those based on elliptic curves.

6.2. Diffie-Hellman Cryptanalysis

Diffie-Hellman key exchange was the first published public-key algorithm [108]. In the simple case of prime groups, Alice and Bob agree on a prime p and a generator g of a multiplicative subgroup modulo p. Alice sends $g^a \bmod p$, Bob sends $g^b \bmod p$, and each computes a shared secret $g^{ab} \bmod p$. While there is also a Diffie-Hellman exchange over elliptic curve groups, we address only the "mod p" case.

The security of Diffie-Hellman is not known to be equivalent to the discrete log problem (except in certain groups[105, 213, 214]), but computing discrete logs remains the best known cryptanalytic attack. An attacker who can find the discrete log x from $y = g^x \bmod p$ can easily find the shared secret.

Textbook descriptions of discrete log can be misleading about the computational tradeoffs, for example by balancing parameters to minimize overall time to compute a *single* discrete

log. In fact, as illustrated in Figure 11, a single large precomputation on p can be used to efficiently break *all* Diffie-Hellman exchanges made with that prime.

The typical case. Diffie-Hellman is typically implemented with prime fields and large group orders. In this case, the most efficient discrete log algorithm is the number field sieve (NFS) [154, 183, 276].[1] There is a closely related number field sieve algorithm for factoring [101, 202], and in fact many parts of the implementations can be shared. The general technique is called index calculus and has four stages with different computational properties. The first three steps are only dependent on the prime p and comprise most of the computation.

First is *polynomial selection*, in which one finds a polynomial $f(z)$ defining a number field $\mathbb{Q}(z)/f(z)$ for the computation. (For our cases, $f(z)$ typically has degree 5 or 6.) This parallelizes well and is only a small portion of the runtime.

In the second stage, *sieving*, one factors ranges of integers and number field elements in batches to find many relations of elements, all of whose prime factors are less than some bound B (called B-smooth). Modern implementations use *special-q* lattice sieving, which for each special q explores a sieving region of 2^{2I} candidates, where I is a parameter. Sieving parallelizes well since each special q is handled independently of the others, but is computationally expensive, because we must search through and attempt to factor many elements. The time for this step depends on heuristic estimates of the probability of encountering B-smooth numbers in this search; it also depends on I and on the number of special q to consider before having enough relations.

In the third stage, *linear algebra*, we construct a large, sparse matrix consisting of the coefficient vectors of prime factorizations we have found. A nonzero kernel vector of the matrix modulo the order q of the group will give us logs of many small elements. This

[1] Recent spectacular advances in discrete log algorithms have resulted in a quasi-polynomial algorithm for small-characteristic fields [47], but these advances are not known to apply to the prime fields used in practice.

database of logs serves as input to the final stage. The difficulty depends on q and the matrix size and can be parallelized in a limited fashion.

The final stage, *descent*, actually deduces the discrete log of the target y. We re-sieve until we can find a set of relations that allow us to write the log of y in terms of the logs in the precomputed database. This step is accomplished in three phases: an initialization phase, which tries to write the target in terms of medium-sized primes, a middle phase, in which these medium-sized primes are further sieved until they can be represented by elements in the database of known logs, and a final phase that actually reconstructs the target using the log database. Crucially, descent is the only NFS stage that involves y (or g), so polynomial selection, sieving, and linear algebra can be done once for a prime p and reused to compute the discrete logs of many targets.

The running time of this algorithm is

$$L_p(1/3, (64/9)^{1/3}) = \exp\left((1.923 + o(1))(\log p)^{1/3}(\log\log p)^{2/3}\right).$$

This is obtained by tuning many parameters, including the degree of f, the sieving region parameter I, and, most importantly, the smoothness bound B. Early articles (e.g. [154]) encountered technical difficulties with descent and reported that the complexity of this step would equal that of the precomputation; this may have contributed to misconceptions about the performance of the NFS for discrete logs. More recent analyses have improved the complexity of descent to $L_p(1/3, 1.442)$ [96], and later to $L_p(1/3, 1.232)$ [46], which is much cheaper than the precomputation in practice.

The numerous parameters of the algorithm allow some flexibility to reduce time on some computational steps at the expense of others. For example, sieving more will result in a smaller matrix, making linear algebra cheaper, and doing more work in the precomputation makes the final descent step easier. In Section 6.3.3, we show how exploiting these tradeoffs allows us to quickly compute 512-bit discrete logs in order to perform an effective man-in-

the-middle attack on TLS.

Improperly generated groups. A different family of algorithms runs in time exponential in group order, and they are practical even for large primes when the group order is small or has many small prime factors. To avoid this, most implementations use "safe" primes, which have the property that $p - 1 = 2q$ for some prime q, so that the only possible subgroups have order 2, q, or $2q$. However, as we show in Section 6.3.5, improperly generated groups are sometimes used in practice and susceptible to attack.

The baby-step giant-step [279] and Pollard rho [261] algorithms both take $\sqrt{q}$ time to compute a discrete log in any (sub)group of order q, while Pollard lambda [261] can find $x < t$ in time $\sqrt{t}$. These parallelize well [299], and precomputation can speed up individual log calculations. If the factorization of the subgroup order q is known, one can use any of the above algorithms to compute the discrete log in each subgroup of order $q_i^{e_i}$ dividing q, and then recover x using the Chinese remainder theorem. This is the Pohlig-Hellman algorithm [259], which costs $\sum_i e_i\sqrt{q_i}$ using baby-step giant-step or Pollard rho.

Standard primes. Generating primes with special properties can be computationally burdensome, so many implementations use fixed or standardized Diffie-Hellman parameters. A prominent example is the Oakley groups [249], which give "safe" primes of length 768 (Oakley Group 1), 1024 (Oakley Group 2), and 1536 (Oakley Group 5). These groups were published in 1998 and have been used for many applications since, including IKE, SSH, Tor, and OTR.

When primes are of sufficient strength, there seems to be no disadvantage to reusing them. However, widespread reuse of Diffie-Hellman groups can convert attacks that are at the limits of an adversary's capabilities into devastating breaks, since it allows the attacker to amortize the cost of discrete log precomputation among vast numbers of potential targets.

Source	Popularity	Prime

Apache	82%	`9fdb8b8a004544f0045f1737d0ba2e0b`
		`274cdf1a9f588218fb435316a16e3741`
		`71fd19d8d8f37c39bf863fd60e3e3006`
		`80a3030c6e4c3757d08f70e6aa871033`

mod_ssl	10%	`d4bcd52406f69b35994b88de5db89682`
		`c8157f62d8f33633ee5772f11f05ab22`
		`d6b5145b9f241e5acc31ff090a4bc711`
		`48976f76795094e71e7903529f5a824b`

(*others*)	8%	(463 distinct primes)

Table 31: **Top 512-bit DH primes for TLS**—8.4% of Alexa Top 1M HTTPS domains allow DHE_EXPORT, of which 92.3% use one of the two most popular primes, shown here.

6.3. Attacking TLS

TLS supports Diffie-Hellman as one of several possible key exchange methods, and about two-thirds of popular HTTPS sites allow it, most commonly using 1024-bit primes. However, a smaller number of servers also support legacy "export-grade" Diffie-Hellman using 512-bit primes that are well within reach of NFS-based cryptanalysis. Furthermore, for both normal and export-grade Diffie-Hellman, the vast majority of servers use a handful of common groups.

In this section, we exploit these facts to construct a novel attack against TLS, which we call the Logjam attack. First, we perform NFS precomputations for the two most popular 512-bit primes on the web, so that we can quickly compute the discrete log for any key-exchange message that uses one of them. Next, we show how a man-in-the-middle, so armed, can attack connections between popular browsers and any server that allows export-grade Diffie-Hellman, by using a TLS protocol flaw to downgrade the connection to export-strength and then recovering the session key. We find that this attack with our precomputations can compromise about 7.8% of HTTPS servers among Alexa Top Million domains.

6.3.1. TLS and Diffie-Hellman

The TLS handshake begins with a negotiation to determine the crypto algorithms used for the session. The client sends a list of supported ciphersuites (and a random nonce cr) within the ClientHello message, where each ciphersuite specifies a key exchange algorithm

and other primitives. The server selects a ciphersuite from the client's list and signals its selection in a ServerHello message (containing a random nonce sr).

TLS specifies ciphersuites supporting multiple varieties of Diffie-Hellman. Textbook Diffie-Hellman with unrestricted strength is called "ephemeral" Diffie-Hellman, or DHE, and is identified by ciphersuites that begin with `TLS_DHE_*`.[2] In DHE, the server is responsible for selecting the Diffie-Hellman parameters. It chooses a group (p, g), computes g^b, and sends a ServerKeyExchange message containing a signature over the tuple (cr, sr, p, g, g^b) using the long-term signing key from its certificate. The client verifies the signature and responds with a ClientKeyExchange message containing g^a.

To ensure agreement on the negotiation messages, and to prevent downgrade attacks [302], each party computes the TLS master secret from g^{ab} and calculates a MAC of its view of the handshake transcript. These MACs are exchanged in a pair of Finished messages and verified by the recipients. Thereafter, client and server start exchanging application data, protected by an authenticated encryption scheme with keys also derived from g^{ab}.

To comply with 1990s-era U.S. export restrictions on cryptography, SSLv3 and TLSv1.0 supported reduced-strength DHE_EXPORT ciphersuites that were restricted to primes no longer than 512 bits. In all other respects, DHE_EXPORT protocol messages are identical to DHE. The relevant export restrictions are no longer in effect, but many libraries and servers maintain support for backwards compatibility. Many TLS servers are still configured with two groups: a strong 1024-bit group for regular DHE key exchanges and a 512-bit group for legacy DHE_EXPORT. This has been considered safe because most modern TLS clients do not offer or accept DHE_EXPORT ciphersuites.

To understand how HTTPS servers in the wild use Diffie-Hellman, we modified the ZMap [114] toolchain to offer DHE and DHE_EXPORT ciphersuites and scanned TCP/443 on both the full public IPv4 address space and the Alexa Top 1M domains. The scans took place in

[2]TLS also supports a rarely used "static" Diffie-Hellman format, where the server's key exchange value is fixed and contained in its certificate. New ciphersuites that use elliptic curve Diffie-Hellman (ECDHE) are gaining in popularity, but we focus exclusively on the traditional prime field variety.

March 2015. Of 539,000 HTTPS sites among Top 1M domains, we found that 68.3% supported DHE and 8.4% supported DHE_EXPORT. Of 14.3 million IPv4 HTTPS servers with browser-trusted certificates, 23.9% supported DHE and 4.9% DHE_EXPORT.

While the TLS protocol allows servers to generate their own Diffie-Hellman parameters, the overwhelming majority use one of a handful of primes. As shown in Table 31, just two 512-bit primes account for 92.3% of Alexa Top 1M domains that support DHE_EXPORT, and 92.5% of all servers with browser-trusted certificates that support DHE_EXPORT. (Non-export DHE follows a similar distribution with longer primes.) The most popular 512-bit prime was hard-coded into many versions of Apache. Introduced in 2005 with Apache 2.1.5, it was used until 2.4.7, which disabled export ciphersuites. We found it in use by about 564,000 servers with browser-trusted certificates. The second most popular 512-bit prime is the default used for DHE_EXPORT when using `mod_ssl`. It was introduced in version 2.3.0 in 1999. We found it in use by about 89,000 servers with browser-trusted certificates.

6.3.2. Active Downgrade to Export-Grade DHE

Given the widespread use of these primes, an attacker with the ability to compute discrete logs in 512-bit groups could efficiently break DHE_EXPORT handshakes for about 8% of Alexa Top 1M HTTPS sites, but modern browsers never negotiate export-grade ciphersuites. To circumvent this, we show how an attacker who can compute 512-bit discrete logs *in real time* can downgrade a regular DHE connection to use a DHE_EXPORT group, and thereby break both the confidentiality and integrity of application data.

The attack, which we call Logjam, is depicted in Figure 12 and relies on a flaw in the way TLS composes DHE and DHE_EXPORT. When a server selects DHE_EXPORT for a handshake, it proceeds by issuing a signed ServerKeyExchange message containing a 512-bit p_{512}, but the structure of this message is identical to the message sent during standard DHE ciphersuites. Critically, the signed portion of the server's message fails to include any indication of the specific ciphersuite that the server has chosen. Provided that a client offers DHE, an active attacker can rewrite the client's ClientHello to offer a corresponding DHE_EXPORT ciphersuite

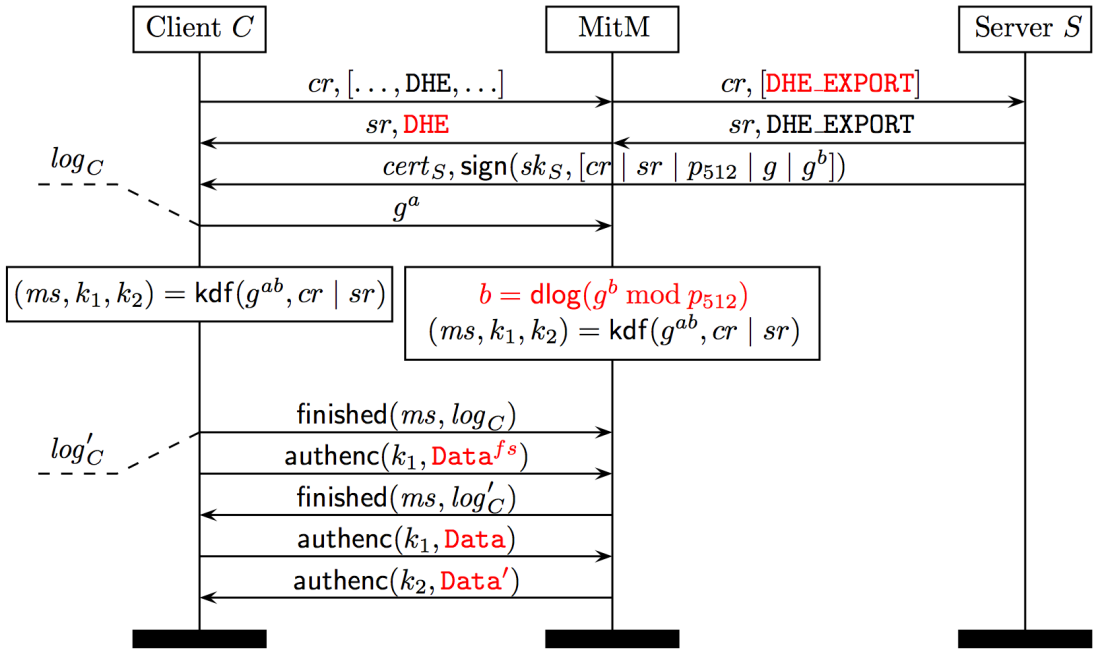


Figure 12: **The Logjam attack**—A man-in-the-middle can force TLS clients to use export-strength DH with any server that allows DHE_EXPORT. Then, by finding the 512-bit discrete log, the attacker can learn the session key and arbitrarily read or modify the contents. $\texttt{Data}^{fs}$ refers to False Start [201] application data that some TLS clients send before receiving the server's Finished message.

accepted by the server and remove other ciphersuites that could be chosen instead. The attacker rewrites the ServerHello response to replace the chosen DHE_EXPORT ciphersuite with a matching non-export ciphersuite and forwards the ServerKeyExchange message to the client as is. The client will interpret the export-grade tuple (p_{512}, g, g^b) as valid DHE parameters chosen by the server and proceed with the handshake. The client and server have different handshake transcripts at this stage, but an attacker who can compute b in close to real time can then derive the master secret and connection keys to complete the handshake with the client, and then freely read and write application data pretending to be the server.

There are two remaining challenges in implementing this active downgrade attack. The first is to compute individual discrete logs in close to real time, and the second is to delay

handshake completion until the discrete log computation has had time to finish. We address these in the next subsections.

Comparison with previous attacks. Logjam is reminiscent of the recent FREAK [61] attack, in which an attacker downgrades a regular RSA key exchange to one that uses export-grade 512-bit ephemeral RSA keys, relying on a bug in several TLS client implementations. The attacker then factors the ephemeral key to hijack future connections that use the same key. The cryptanalysis takes several hours on commodity hardware and is usable until the server generates a fresh ephemeral RSA key (typically when it restarts).

In contrast, Logjam is due to a protocol flaw in TLS, not an implementation bug. From a client perspective, the only defense is to reject small primes in DHE handshakes. (Prior to this work, most popular browsers accepted p of size ≥ 512 bits.) Logjam affects fewer servers than FREAK, but, as we shall see, the cost per compromised connection is far lower, since the precomputation for each 512-bit group can be used indefinitely against all servers that use that group, and since each individual discrete log only takes about a minute.

Logjam and FREAK both follow the same pattern as other cross-protocol attacks discovered in TLS. As early as SSL 3.0, Schneier and Wagner noted a related vulnerability that they called *key exchange rollback* [302]. Mavrogiannopoulos et al. showed how explicit-curve ECDHE handshakes could be confused with DHE handshakes [215]. All these attacks could be prevented by additionally signing the ciphersuite in the ServerKeyExchange message. We expect that TLSv1.3 will fix this protocol flaw. More generally, Logjam can also be interpreted as a backwards compatibility attack [178] where one party uses only strong cryptography but the other supports both strong and weak ciphersuites.

6.3.3. 512-bit Discrete Log Computations

We modified CADO-NFS [45] to implement the number field sieve discrete log algorithm from Section 6.2 and applied it to three 512-bit primes, including the top two DHE_EXPORT primes shown in Table 31. Precomputation took 7 days for each prime, after which computing individual logs took a median of 70 seconds. We list the runtime for each stage of

the computation below. The times were about the same for each prime.

Precomputation. As illustrated in Figure 11, the precomputation phase includes the polynomial selection, sieving, and linear algebra steps. For this precomputation, we deliberately sieved more than strictly necessary. This enabled two optimizations: first, with more relations obtained from sieving, we eventually obtain a larger database of known logs, which makes the descent faster. Second, more sieving relations also yield a smaller linear algebra step, which is desirable because sieving is much easier to parallelize than linear algebra.

For the polynomial selection and sieving steps, we used idle time on 2000–3000 CPU cores in parallel, of which most CPUs were Intel Sandy Bridge. Polynomial selection ran for about 3 hours, which in total corresponds to 7,600 core-hours. Sieving ran for 15 hours, corresponding to 21,400 core-hours. This sufficed to collect 40,003,519 relations of which 28,372,442 were unique, involving 15,207,865 primes of at most 27 bits (hence bound B from Section 6.2 is 2^{27}).

From this data set, we obtained a square matrix with 2,157,378 rows and columns, with 113 nonzero coefficients per row on average. We solved the corresponding linear system on a 36-node cluster with two 8-core Intel Xeon E5-2650 CPUs per node, connected with Infiniband FDR. We used the block Wiedemann algorithm [97, 291] with parameters $m = 18$ and $n = 6$. Using the unoptimized implementation from CADO-NFS [45] for linear algebra over GF(p), the computation finished in 120 hours, corresponding to 60,000 core-hours. We expect that optimizations could bring this cost down by at least a factor of three.

In total, the wall-clock time for each precomputation was slightly over one week. Each resulting database of known logs for the descent occupies about 2.5 GB in ASCII format.

Descent. Once this precomputation was finished, we were able to run the final descent step to compute individual discrete logs in about a minute for targets in each of these groups. In order to save time on individual computations, we implemented a client-server architecture using the ZeroMQ messaging library. The server maintains the precomputed

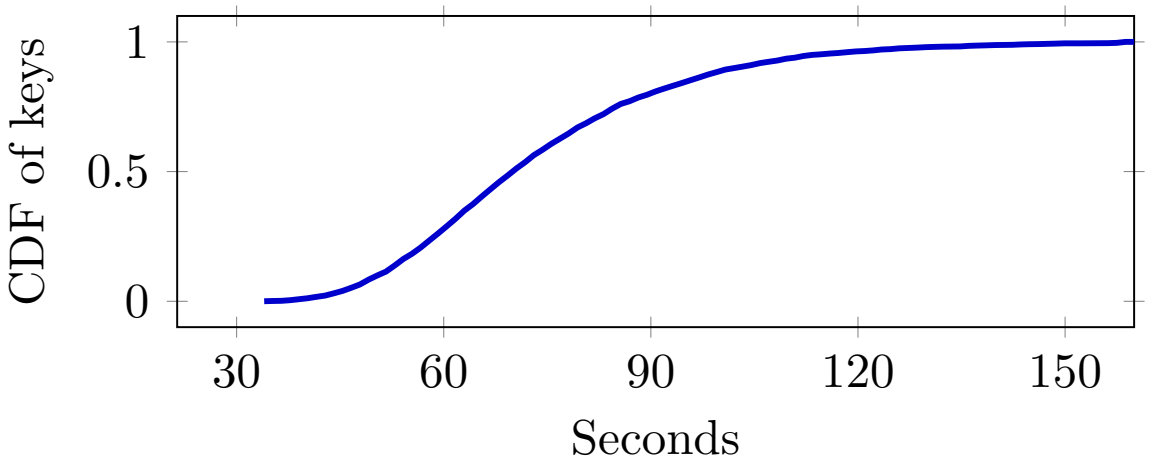


Figure 13: **Individual discrete log time for 512-bit DH**—After a week-long precomputation for each of the two top export-grade primes (see Table 31), we can quickly break any key exchange that uses them. Here we show times for computing 3,500 individual logs; the median is 70 seconds.

data in RAM and returns logs for values passed to it by clients.

We implemented the descent calculation in a mix of Python and C. The first and second stages are parallelized and run sieving in C, and the final discrete log is deduced in Python. We ran the server on a machine with two 18-core Intel Xeon E5-2699 CPUs and 128 GB of RAM. On average, computing individual logs took about 70 seconds, but the time varied from 34 to 206 seconds (see Fig. 13). This is divided between about 20 seconds for descent initialization and the remainder on the middle phase. Further optimizations—such as more effective parallelization on the middle phase or additional sieving—should bring the median time well below a minute.

For purposes of comparison, a single 512-bit RSA factorization using the CADO-NFS implementation takes about eight days of wall-clock time on the computer used for the descent, and about three hours parallelized across 1,800 cores of Amazon EC2 `c4.8xlarge` instances.

6.3.4. Active Attack Implementation

We implemented a man-in-the-middle network attacker that sits between a TLS client (web browser) and any server that supports DHE_EXPORT and uses the most common 512-

bit Apache group. Our implementation follows the message sequence in Figure 12: it downgrades the connection towards the server, computes the session keys, and takes over the connection towards the client by impersonating the server.

The main challenge is to compute the shared secret g^{ab} before the handshake completes in order to forge a Finished message from the server. With our descent implementation, the computation takes an average of 70 seconds, but there are several ways an attacker can work around this delay:

Non-browser clients. Different TLS clients impose different time limits for the handshake, after which they kill the connection. Command-line clients such as `curl` and `git` often run unattended, so they have long or no timeouts, and we can hijack their connections without difficulty.

TLS warning alerts. Web browsers tend to have shorter timeouts, but we can keep their connections alive by sending TLS warning alerts, which are ignored by the browser but reset the handshake timer. For example, this allows us to keep Firefox's TLS connections alive indefinitely. (Other browsers we tested close the connection after a minute.) Although the victim connection still takes much longer than usual, the attacker might choose to compromise a request for a background resource that does not delay rendering the page.

Ephemeral key caching. Many TLS servers do not use a fresh value b for each connection, but instead compute g^b once and reuse it for multiple negotiations. Without enabling the `SSL_OP_SINGLE_DH_USE` option, OpenSSL will reuse g^b for the lifetime of a TLS context. While both Apache and Nginx internally apply this option, certain load balancers, such as stud [24], do not. The F5 BIG-IP load balancers and hardware TLS frontends will reuse g^b unless the "Single DH" option is checked [303]. Microsoft Schannel caches g^b for two hours—this setting is hard-coded. For these servers, an attacker can compute the discrete log of g^b from one connection and use it to attack later handshakes, avoiding the need to do the computation online. By randomly sampling IPv4 hosts serving browser-trusted

certificates that support DHE, we found that 17% reused g^b at least once over the course of 20 handshakes, and that 15% only used one value. However, for DHE_EXPORT, only 0.1% reused g^b, likely because Microsoft IIS does not support 512-bit export ciphersuites.

TLS False Start. Even when clients enforce shorter timeouts and servers do not reuse values for b, the attacker can still break the confidentiality of user requests if the client supports the TLS False Start extension [201]. This extension reduces connection latency by having the client send early application data (such as an HTTP request) without waiting for the server's Finished message to arrive. Recent versions of Chrome, Internet Explorer, and Firefox implement False Start, but their policies on when to enable it vary between versions. Firefox 35, Chrome 41, and Internet Explorer (Windows 10) send False Start data with DHE. In these cases, a man-in-the-middle can record the handshake and decrypt the False Start payload at leisure. We note that this initial data sent by a browser often contains sensitive user authentication information, such as passwords and cookies.

6.3.5. Other Weak and Misconfigured Groups

In our scans, we found several other exploitable security issues in the DHE configurations used by TLS servers.

512-bit primes in non-export DHE. We found 2,631 servers with browser-trusted certificates (and 118 in the Top 1M domains) that used 512-bit or weaker primes for non-export DHE. In these instances, active attacks may be unnecessary. If a browser negotiates a DHE ciphersuite with one of these servers, a *passive* eavesdropper can later compute the discrete log and obtain the TLS session keys for the connection. An active attack may still be necessary when the client's ordering of ciphersuites would result in the server not selecting DHE. In this case, as in the DHE_EXPORT downgrade attack, an active attacker can force the server to choose a vulnerable DHE ciphersuite.

As a proof-of-concept, we implemented a passive eavesdropper for regular DHE connections and used it to decrypt test connections to `www.fbi.gov`. Until April 2015, this server used the default 512-bit DH group from OpenSSL, which was the third group for which we

performed the NFS precomputation. The website no longer supports DHE.

Attacks on composite-order subgroups. Failure to generate Diffie-Hellman primes according to best practices can result in devastating attacks. Not every TLS server uses "safe" primes. Out of approximately 70,000 distinct primes seen across both export and non-export TLS scans, 4,800 were not safe, meaning that $(p-1)/2$ was composite. (Incidentally, we also found 9 composite p.) These groups are not necessarily vulnerable, as long as g generates a group with at least one sufficiently large subgroup order to rule out the Pohlig-Hellman algorithm as an attack.

In some real-life configurations, however, choosing such primes can lead to an attack. For efficiency reasons, some implementations use ephemeral keys g^x with a short exponent x; commonly suggested sizes for x are as small as 160 or 224 bits, intended to match the estimated strength of a 1024- or 2048-bit group. For safe p, such exponent lengths are not known to decrease security, as the most efficient attack will be the Pollard lambda algorithm. But if the order of the subgroup generated by g has small factors, they can be used to recover information about exponents. From a subset of factors $\{q_1^{e_1} \dots q_k^{e_k}\}$ with $\prod_i q_i^{e_i} = z$, Pohlig-Hellman can recover $x \mod z$ in time $\sum_i e_i \sqrt{q_i}$. If $x \leq z$, this suffices to recover x. If not, Pollard lambda can use this information to recover x in time $\sqrt{x/z}$. This attack was first described as hypothetical by van Oorschot and Wiener [300].

To see if TLS servers in the wild were vulnerable to this attack, we tested various non-safe primes found in our scans. For each non-safe prime p, we opportunistically factored $p-1$ using Bernstein's batch method [53]. We then ran the GMP-ECM implementations of the Pollard $p-1$ algorithm and the ECM factoring methods [318] for 5 days parallelized across 28 cores and discovered 36,447 prime factors.

We then examined the generators g used with each prime p. We classified a tuple (p, g, y) sent by a server as interesting if the prime factorization of $p-1$ had revealed prime factors of the order of g, and ordered them by the estimated work required using Pohlig-Hellman

and Pollard lambda to recover a target private exponent x of length ranging from 64 to 256 bits. There were 753 (p, g) pairs where we knew factors of the subgroup generated by g; these had been used for 40,903 connections across all of our scans.

We implemented the van Oorschot and Wiener algorithm in Sage [286] using a parallel Pollard rho implementation that we wrote in C using the GMP library. We used the distinguished points method for collision detection; for a prime known in advance, this implementation can be arbitrarily sped up by precomputing a table of distinguished points.

We computed partial information about the server secret exponent used in 460 exchanges and were able to recover the whole exponent used by 159 different hosts, 53 of which authenticated with valid browser-trusted certificates. In all cases, the vulnerable hosts used 512-bit prime moduli; three of them used 160-bit exponents and the rest used 128 bits. The order of the largest-order subgroup ranged from 46 bits (which finishes in seconds) to 81 bits (which took between 50 and 176 hours) implementation. The Pollard lambda calculations used interval width varying from 40 to 70 bits.

Our computations would have allowed us to hijack connections to a variety of vulnerable TLS servers, including web interfaces for VPN devices (48 hosts), communications software (21 hosts), web conferencing servers (27 hosts), and FTP servers (6 hosts). As a proof-of-concept, we modified our man-in-the-middle attacker of Section 6.3.3 to impersonate a vulnerable server and capture user credentials. Compared to an attack using NFS, we could compute the discrete log with a delay hardly noticeable for browser users.

Misconfigured groups. The Digital Signature Algorithm (DSA) [239] uses primes p such that $p - 1$ has a large prime factor q and g generates only a subgroup of order q. When using properly generated DSA parameters, these groups are secure for use in Diffie-Hellman key exchanges. Notably, DSA groups are hard-coded in Java's `sun.security.provider` package and are used by default in many Java-based TLS servers. However, some servers in our scans used Java's DSA primes as p but mistakenly used the DSA group order q in

	Sieving			Linear Algebra		Descent	
	I	$\log_2 B$	core-yrs	rows	core-yrs	core-time	
RSA-512	14	29	0.5	4.3M	0.33		Timings with default CADO-NFS parameters.
DH-512	15	27	2.5	2.1M	7.7	10 mins	For the computations in this paper; may be suboptimal.
RSA-768	16	37	800	250M	100		Est. based on [195] with less sieving.
DH-768	17	35	8K	150M	28.5K	2 days	Est. based on [75, 195] and our own experiments.
RSA-1024	18	42	1M	8.7B	120K		Est. based on complexity formula.
DH-1024	19	40	10M	5.2B	35M	30 days	Est. based on complexity formula and our experiments.

Table 32: **Estimating costs for factoring and discrete log**—For sieving, we give two important parameters: the number of bits of the smoothness bound B and the sieving region parameter I. For linear algebra, all costs for DH are for safe primes; for DSA primes with q of 160 bits, this should be divided by 6.4 for 1024 bits, 4.8 for 768 bits, and 3.2 for 512 bits.

the place of the generator g. We found 5,741 hosts misconfigured this way.

This substitution of q for g is likely due to a usability problem: the canonical ASN.1 representation of Diffie-Hellman key exchange parameters (coming from PKCS#3) is a sequence (p, g), while that of DSA parameters (coming from PKIX) is (p, q, g); we conjecture that the confusion between these formats led to a simple programming error.

In a DSA group, the subgroup generated by q is likely to have many small prime factors in its order, since for p generated according to [239], $(p-1)/q$ is a random integer. For Java's `sun.security.provider` 512-bit prime, using q as a generator leaks 290 bits of information about exponents at a cost of roughly 2^{40} operations. Luckily, since the provider generates exponents of length $\max(n/2, 384)$ for n-bit p, this does not suffice to recover a full exponent. Still, this misconfiguration bug results in a significant loss of security and serves as a cautionary tale for programmers.

6.4. State-Level Threats to DH

The previous sections demonstrate the existence of practical attacks against Diffie-Hellman key exchange as currently used by TLS. However, these attacks rely on the ability to downgrade connections to export-grade crypto or on the use of unsafe parameters. In this section we address the following question: how secure is Diffie-Hellman in broader practice, as used in other protocols that do not suffer from downgrade, and when applied with stronger groups?

To answer this question we must first examine how the number field sieve for discrete log scales to 768- and 1024-bit groups. As we argue below, 768-bit groups, which are still in relatively widespread use, are now within reach for academic computational resources, and performing precomputations for a small number of 1024-bit groups is plausibly within the resources of state-level attackers. The precomputation would likely require special-purpose hardware, but would not require any major algorithmic improvements beyond what is known in the academic literature. We further show that even in the 1024-bit case, the descent time—necessary to solve any specific discrete log instance within a common group—would be fast enough to break individual key exchanges in close to real time.

In light of these results, we examine several standard Internet security protocols—IKE, SSH, and TLS—to determine the vulnerability of their key exchanges to attacks by resourceful attackers. Although the cost of the precomputation for a 1024-bit group is several times higher than for an RSA key of equal size, we observe that a one-time investment could be used to attack millions of hosts, due to widespread reuse of the most common Diffie-Hellman parameters. Unfortunately, our measurements also indicate that it may be very difficult to sunset the use of fixed 1024-bit Diffie-Hellman groups that have long been embedded in standards and implementations.

Finally, we apply this new understanding to a set of recently published documents leaked by Edward Snowden [282] to evaluate the hypothesis that the National Security Agency has *already* implemented such a capability. We show that this hypothesis is consistent with the published details of the intelligence community's cryptanalytic capabilities, and, indeed, matches the known capabilities more closely than other proposed explanations, such as novel breaks on RC4 or AES. We believe that this analysis may help shed light on unanswered questions about how NSA may be gaining access to VPN, SSH, and TLS traffic.

6.4.1. Scaling NFS to 768- and 1024-bit DH

Estimating the cost for discrete log cryptanalysis at longer key sizes is far from straightforward, due in part to the complexity of parameter tuning and to tradeoffs between the

sieving and linear algebra steps, which have very different computational characteristics. (Much more attention has gone to understanding 1024-bit factorization, but, even there, many published estimates are crude extrapolations of the asymptotic complexity.) We attempt estimates for 768- and 1024-bit discrete log based on the existing literature and our own experiments, but further work is needed for greater confidence, particularly for the 1024-bit case. We summarize all the costs, measured or estimated, in Table 32.

DH-768: Feasible with academic power. For the 768-bit case, we base our estimates on the recent discrete log record at 596 bits [75] and the integer factorization record of 768 bits from 2009 [195]. While the algorithms for factorization and discrete log are similar, the discrete log linear algebra stage is many times more difficult, as the matrix entries are no longer Boolean. We can reduce overall time by sieving more, thus generating a smaller input matrix to the linear algebra step. Since sieving parallelizes better than linear algebra, this tradeoff is desirable for large inputs.

A 596-bit factorization takes about 5 core-years, most of it spent on sieving. In comparison, the record 596-bit discrete log effort tuned parameters such that they spent 50 core-years on sieving. This reduced their linear algebra calculation to 80 core-years. We used this same strategy in our 512-bit experiments in Section 6.3.3.

Similarly, the 768-bit RSA factoring record spent more time on sieving in order to save time on the linear algebra step. The cost of sieving was around 1500 core-years, and the matrix that was produced had 200M rows and columns. As a result, the linear algebra took 150 core-years, but taking algorithmic improvements since 2009 into account and optimizing for the total time,[3] we estimate that factoring an RSA-768 integer would take 900 core-years in total.

For a 768-bit discrete log, we can expect that ten times as much sieving as the RSA case would reduce the matrix to around 150M rows. We extrapolate from experiments with

[3] We would lower the smoothness bounds compared to the parameters in [195].

existing software that this linear algebra would take 28,500 core-years, for a total of 36,500 core-years. This is within reach by computing power available to academics.

The descent step takes relatively little time. We experimented with both CADO-NFS and a new implementation with GMP-ECM based on the early-abort strategy described in [57]. Using these techniques, the initial descent phase took an average of around 1 core-day. The remaining phase uses sieving much as in the precomputation; extrapolating from experiments, the rest of the descent should take at most 1 core-day. In total, after precomputation, the cost of a single 768-bit discrete log computation is around 2 core-days and is easily parallelizable.

DH-1024: Plausible with state-level resources. Experimentally extrapolating sieving parameters to the 1024-bit case is difficult due to the tradeoffs between the steps of the algorithm and their relative parallelism. The prior work proposing parameters for factoring a 1024-bit RSA key is thin: [194] proposes smoothness bounds of 42 bits, but the proposed value of the sieving region parameter `I` is clearly too small, giving too few smooth results per sieving subtask. Since no publicly available software can currently deal with values of `I` larger than those proposed, we could not experimentally update the estimates of this paper with more relevant parameter choices.

Without better parameter choices, we resort to extrapolating from asymptotic complexity. For the number field sieve, the complexity is $\exp\left((k+o(1))(\log N)^{1/3}(\log\log N)^{2/3}\right)$, where N is the integer to factor or the prime modulus for discrete log, and k is an algorithm-specific constant. This formula is inherently imprecise, since the $o(1)$ in the exponent can hide polynomial factors. This complexity formula, with $k = 1.923$, describes the overall time for both discrete log and factorization, which are both dominated by sieving and linear algebra in the precomputation. The space complexity (the size of the matrix in memory) is the square root of this function, i.e., the same function, taking $k = 0.9615$. Discrete log descent has a complexity of the same form as well; [46, Chapter 4] gives $k = 1.232$, using an early-abort strategy similar to the one in [57] mentioned above.

Evaluating the formula for 768- and 1024-bit N gives us estimated multiplicative factors by which time and space will increase from the 768- to the 1024-bit case. For precomputation, the total time complexity will increase by a factor of 1220, while space complexity will increase by a factor of 35. These are valid for both factorization and discrete log, since they have the same asymptotic behavior. Hence, for DH-1024, we get a total cost for the precomputation of about 45M core-years. The time complexity for each individual log after the precomputation should be multiplied by 95. This last number does not correspond to what we observed in practice; we attribute that to the fact that the descent step has been far less studied both in theory and in practice compared to the other steps.

For 1024-bit descent, we experimented with our early-abort implementation to inform our estimates for descent initialization, which should dominate the individual discrete log computation. For a random target in Oakley Group 2, initialization took 22 core-days, yielding a few primes of at most 130 bits to be descended further. In twice this time, we reached primes of about 110 bits. At this point, we were certain to have bootstrapped the descent, and could continue down to the smoothness bound in a few more core-days if proper sieving software were available. Thus we estimate that a 1024-bit descent would take about 30 core-days, once again easily parallelizable.

Costs in hardware. Although 45M core-years is a huge computational effort, it is not necessarily out of reach for a nation state. Moreover, at this scale, significant cost savings could be realized by developing application-specific hardware.

Sieving is a natural target for hardware implementation. To our knowledge, the best prior description of an ASIC implementation of 1024-bit sieving is the 2007 work of Geiselmann and Steinwandt [137]. In the following, we update their estimates for modern techniques and adjust parameters for discrete log. We increase their chip count by a factor of ten to sieve more and save on linear algebra as above, giving an estimate of 3M chips to complete sieving in one year. Shrinking the dies from the 130 nm technology node used in the paper to a more modern size reduces costs, as transistors are cheaper at newer technologies. With

standard transistor costs and utilization, this would cost about $2 per chip to manufacture, after fixed design and tape-out costs of roughly $2M [208]. This suggests that an $8M investment would buy enough ASICs to complete the DH-1024 sieving precomputation in one year. Since a step of descent uses sieving, the same hardware could likely be reused to speed calculations of individual logs.

Estimating the financial cost for the linear algebra is more difficult, since there has been little work on designing chips that are suitable for the larger fields involved in discrete log. To derive a rough estimate, we can begin with general purpose hardware and the core-year estimate from Table 32. The Titan supercomputer [240]—at 300,000 CPU cores, currently the most powerful supercomputer in the U.S.—would take 117 years to complete the 1024-bit linear algebra stage. Titan was constructed in 2012 for $94M, suggesting a cost of $11B in supercomputers to finish this step in a year. In the context of factorization, moving linear algebra from general purpose CPUs to ASICs has been estimated to reduce costs by a factor of 80 [138]. If we optimistically assume that a similar reduction can be achieved for discrete log, the hardware cost to perform the linear algebra for DH-1024 in one year is plausibly on the order of hundreds of millions of dollars.

To put this dollar figure in context, the FY 2012 budget for the U.S. Consolidated Cryptologic Program (which includes the NSA) was $10.5 billion[4] [1]. The agency's classified 2013 budget request, which prioritized investment in "groundbreaking cryptanalytic capabilities to defeat adversarial cryptography and exploit internet traffic," included notable $100M increases in two programs [1]: "cryptanalytic IT services" (to $247M), and a cryptically named "cryptanalysis and exploitation services program C" (to $360M). NSA's leaked strategic plan for the period called for it to "continue to invest in the industrial base and drive the state of the art for high performance computing to maintain pre-eminent cryptanalytic capability for the nation" [19].

[4]The National Science Foundation's budget was $7 billion.

6.4.2. Is NSA Breaking 1024-bit DH?

Our calculations suggest that it is plausibly within NSA's resources to have performed number field sieve precomputations for at least a small number of 1024-bit Diffie-Hellman groups. This would allow them to break any key exchanges made with those groups in close to real time. If true, this would answer one of the major cryptographic questions raised by the Edward Snowden leaks: How is NSA defeating the encryption for widely used VPN protocols?

Classified documents published by Der Spiegel [282] indicate that NSA is passively decrypting IPsec connections at significant scale. The documents do not describe the cryptanalytic techniques used, but they do provide an overview of the attack system architecture. After reviewing how IPsec key establishment works, we will use the published information to evaluate the hypothesis that the NSA is leveraging precomputation to calculate discrete logs at scale.

IKE. Internet Key Exchange (IKE) is the main key establishment protocol used for IPsec VPNs. There are two versions, IKEv1 [164] and IKEv2 [188], which differ in message structure but are conceptually similar. For the sake of brevity, we will use IKEv1 terminology.

Each IKE session begins with a Phase 1 handshake, in which the client and server select a Diffie-Hellman group from a small set of standardized parameters and perform a key exchange to establish a shared secret. The shared secret is combined with other cleartext values transmitted by each side, such as nonces and cookies, to derive a value called SKEYID. IKE provides several authentication mechanisms, including symmetric pre-shared keys (PSK); when IKEv1 is authenticated with a PSK, this value is incorporated into the derivation of SKEYID.

The resulting SKEYID is used to encrypt and authenticate a Phase 2 handshake. Phase 2 establishes the parameters and key material, KEYMAT, for a cryptographic transport protocol used to protect subsequent traffic, such as Encapsulating Security Payload (ESP) [192] or

Authenticated Header (AH) [191]. In some circumstances, this phase includes an additional round of Diffie-Hellman. Ultimately, KEYMAT is derived from SKEYID, additional nonces, and the result of the optional Phase 2 Diffie-Hellman exchange.

NSA's VPN exploitation process. The documents published by Der Spiegel describe a system named TURMOIL that is used to collect and decrypt VPN traffic. The evidence indicates that this decryption is performed using passive eavesdropping and does not require message injection or man-in-the-middle attacks on IPsec or IKE. Figure 14, an excerpt from one of the documents [20], illustrates the flow of information through the TURMOIL system

The initial phases of the attack involve collecting IKE and ESP payloads and determining whether the traffic matches any tasked selector [7]. If so, TURMOIL transmits the complete IKE handshake and may transmit a small amount of ESP ciphertext to NSA's Cryptanalysis and Exploitation Services (CES) [7, 14] via a secure tunnel. Within CES, a specialized VPN Attack Orchestrator (VAO) system manages a collection of high-performance grid computing resources located at NSA Headquarters and in a data center at Oak Ridge National Laboratory, which perform the computation required to generate the ESP session key [10, 15, 20]. VAO also maintains a database, CORALREEF, that stores cryptographic values, including a set of known PSKs and the resulting "recovered" ESP session keys [15, 20, 23].

The ESP traffic itself is buffered for up to 15 minutes [12], until CES can respond with the recovered ESP keys if they were generated correctly. Once keys have been returned, the ESP traffic is decrypted via hardware accelerators [6] or in software [9, 13]. From this point, decrypted VPN traffic is reinjected into TURMOIL processing infrastructure and passed to other systems for storage and analysis [13]. The documents indicate that NSA is recovering ESP keys at large scale, with a target of 100,000 per hour [12].

Evidence for a discrete log attack. While the ability to decrypt VPN traffic does not by itself indicate a defeat of Diffie-Hellman, there are several features of IKE and the VAO's operation that support this hypothesis.

The IKE protocol has been extensively analyzed [87, 217], and is not believed to be exploitable in standard configurations under passive eavesdropping attacks. In order to recover the session keys for the ESP or AH protocols, the attacker must at minimum recover the SKEYID generated by the Phase 1 exchange. Absent a vulnerability in the key derivation function or transport encryption, this requires the attacker to recover a Diffie-Hellman shared secret after passively observing an IKE handshake.

While IKE is designed to support a range of Diffie-Hellman groups, our Internet-wide scans (Section 6.4.3) show that the vast majority of IKE systems select one particular 1024-bit DH group, Oakley Group 2, even when offered stronger groups.

Given an efficient oracle for solving the discrete logarithm problem, attacks on IKE are possible provided that the attacker can obtain the following: (1) a complete two-sided IKE transcript, including the Diffie-Hellman ephemeral keys g^a and g^b as well as the nonces and cookies transmitted by both sides of the connection, and (2) in IKEv1 only, the PSK used in deriving SKEYID.

Both of the above requirements are also present in the NSA's VPN attack system. As Figure 14 illustrates, a hard requirement of the VAO is the need to obtain the *complete* two-sided IKE transcript [23]. The published documents indicate that this requirement substantially increases the complexity of the attack execution, since IKE transcripts must be reassembled ("paired") whenever the interaction traverses multiple network paths [8, 14, 21, 22].

The attack system also seems to require knowledge of the PSK. Several documents describe techniques for analysts to locate a PSK, including using a database of router configurations [11, 16], the CORALREEF database of known PSKs [23], previously decrypted SSH traffic [23], or system administrator "chatter" [11]. Additionally, NSA is willing to "[r]un attacks to recover PSK" [23].

Of course, this explanation is not dispositive. The possibility remains that NSA could de-

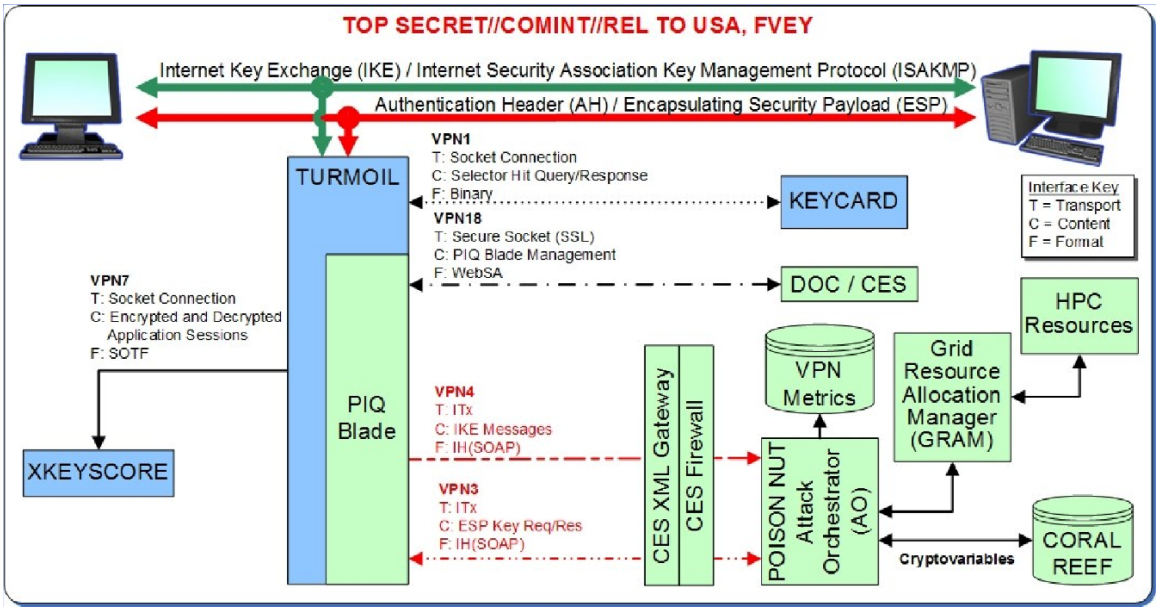


Figure 14: **NSA's VPN decryption infrastructure**—This classified illustration published by Der Spiegel [20] shows captured IKE handshake messages being passed to a high-performance computing system, which returns the symmetric keys for ESP session traffic. The details of this attack are consistent with an efficient break for 1024-bit Diffie-Hellman.

feat IPsec using alternative means. Certain published NSA documents refer to software "implants" on VPN devices, indicating that the use of targeted malware is a piece of the collection strategy [23]; however, the same documents also note that decryption of the resulting traffic *does not* require IKE handshakes, and thus appears to be an alternative mechanism to the VAO attack described above. The most compelling argument for a pure cryptographic attack is the generality of the VAO approach, which appears to succeed across a broad swath of non-compromised devices.

6.4.3. Effects of a 1024-bit Break

In this section, we use Internet-wide scanning to assess the impact of a hypothetical DH-1024 break on three popular protocols: IKE, SSH, and HTTPS. Our measurements indicate that these protocols, as they are commonly used, would be subject to widespread compromise by a state-level attacker who had the resources to invest in precomputation for a small number of common 1024-bit groups.

IKE. We measured how IPsec VPNs use Diffie-Hellman in practice by scanning a 1%

	Vulnerable servers, if the attacker can precompute for ...			
	all 512-bit groups	all 768-bit groups	one 1024-bit group	ten 1024-bit groups
HTTPS Top 1M w/ active downgrade	45,100 (8.4%)	45,100 (8.4%)	205,000 (37.1%)	309,000 (56.1%)
HTTPS Top 1M	118 (0.0%)	407 (0.1%)	98,500 (17.9%)	132,000 (24.0%)
HTTPS Trusted w/ active downgrade	489,000 (3.4%)	556,000 (3.9%)	1,840,000 (12.8%)	3,410,000 (23.8%)
HTTPS Trusted	1,000 (0.0%)	46,700 (0.3%)	939,000 (6.56%)	1,430,000 (10.0%)
IKEv1 IPv4	–	64,700 (2.6%)	1,690,000 (66.1%)	1,690,000 (66.1%)
IKEv2 IPv4	–	66,000 (5.8%)	726,000 (63.9%)	726,000 (63.9%)
SSH IPv4	–	–	3,600,000 (25.7%)	3,600,000 (25.7%)

Table 33: **Estimated impact of Diffie-Hellman attacks**—We use Internet-wide scanning to estimate the number of real-world servers for which typical connections could be compromised by attackers with various levels of computational resources. For HTTPS, we provide figures with and without downgrade attacks on the chosen ciphersuite. All others are passive attacks.

random sample of the public IPv4 address space for IKEv1 and IKEv2 (the protocols used to initiate an IPsec VPN connection) in May 2015. We used the ZMap UDP probe module to measure support for Oakley Groups 1 and 2 (two popular 768- and 1024-bit, built-in groups) and which group servers prefer. To test support for individual groups, we offered only the single group in question. To detect default behavior, we offered servers a variety of DH groups, with the lowest priority groups being Oakley Groups 1 and 2. When measuring server preference, we scanned with the 3DES symmetric cipher—the most commonly supported symmetric cipher in our single group scans. Because of this, the percentages we present for IKEv1 and IKEv2 are a lower bound for the number of servers that prefer Oakley Groups 1 and 2.

Of the 80K hosts that responded with a valid IKE packet, 44.2% were willing to accept an offered proposal from at least one scan. The majority of the remaining hosts responded with a `NO-PROPOSAL-CHOSEN` message regardless of our proposal. Many of these may be site-to-site VPNs that reject our source address. We consider these hosts "unprofiled" and omit them from the results here.

We found that 31.8% of IKEv1 and 19.7% of IKEv2 servers support Oakley Group 1 (768-bit) while 86.1% and 91.0% respectively supported Oakley Group 2 (1024-bit). In our sample of IKEv1 servers, 2.6% of profiled servers preferred the 768-bit Oakley Group 1—which is within cryptanalytic reach today for moderately resourced attackers—and 66.1%

preferred the 1024-bit Oakley Group 2. For IKEv2, 5.8% of profiled servers chose Oakley Group 1, and 63.9% chose Oakley Group 2. This coincides with our anecdotal findings that most VPN clients only offer Oakley Group 2 by default.

SSH. All SSH handshakes complete either a finite field Diffie-Hellman or elliptic curve Diffie-Hellman exchange as part of the SSH key exchange. The SSH protocol explicitly defines support for Oakley Group 2 (1024-bit) and Oakley Group 14 (2048-bit) but also allows a server-defined group, which can be negotiated through an auxiliary Diffie-Hellman Group Exchange (DH-GEX) handshake [131].

In order to measure how SSH uses DH in practice, we implemented the SSH protocol in the ZMap toolchain and scanned 1% random samples of the public IPv4 address space in April 2015. We find that 98.9% of SSH servers support the 1024-bit Oakley Group 2, 77.6% support the 2048-bit Oakley Group 14, and 68.7% support DH-GEX.

During the SSH handshake, the client and server select the client's highest priority mutually supported key exchange algorithm. Therefore, we cannot directly measure what algorithm servers will prefer in practice. In order to estimate this, we performed a scan in which we mimicked the algorithms offered by OpenSSH 6.6.1p1, the latest version of OpenSSH. In this scan, 21.8% of servers preferred the 1024-bit Oakley Group 2, and 37.4% preferred a server-defined group. 10% of the server-defined groups were 1024-bit, but, of those, near all provided Oakley Group 2 rather than a custom group.

Combining these equivalent choices, we find that a state-level attacker who performed NFS precomputations for the 1024-bit Oakley Group 2 (which has been in standards for almost two decades) could passively eavesdrop on connections to 3.6M (25.7%) publicly accessible SSH servers.

HTTPS. DHE is commonly deployed on web servers. 68.3% of Alexa Top 1M sites support DHE, as do 23.9% of sites with browser-trusted certificates. Of the Top 1M sites that support DHE, 84% use a 1024-bit or smaller group, with 94% of these using one of five groups.

Despite widespread support for DHE, a passive eavesdropper can only decrypt connections that organically agree to use Diffie-Hellman. We can estimate the number of sites for which this will occur by offering the same sets of ciphersuites as Chrome, Firefox, and Safari. While the offered ciphers differ slightly between browsers, this turns out to result in negligible differences in whether DHE is chosen.

Approximately 24.0% of browser connections with HTTPS-enabled Top 1M sites (and 10% with browser-trusted sites) will negotiate DHE with one of the ten most popular 1024-bit primes; 17.9% of connections with Top 1M sites could be passively eavesdropped given the precomputation for a single 1024-bit prime. The most popular site that negotiates a DHE ciphersuite using one of the two most common 1024-bit primes is sohu.com (ranked 31st globally).

Mail. TLS is also used to secure email transport. SMTP, the protocol used to relay messages between mail servers, allows a connection to be upgraded to TLS by issuing the `STARTTLS` command. POP3S and IMAPS, used by end users to fetch received mail, wrap the entire connection in TLS.

We studied 1% samples of the public IPv4 address space for IMAPS, POP3S, and SMTP-+StartTLS. We found that 50.7% of SMTP servers supported `STARTTLS`, 41.4% supported DHE, and 14.8% supported DHE_EXPORT ciphers. 15.5% of SMTP servers used one of the ten most common 1024-bit groups.

For IMAPS, 8.4% of servers supported DHE_EXPORT and 75% supported DHE. However, the ten most common 1024-bit primes account for only 5.4% of servers. POP3S deployment is similar, with 8.9% of servers supporting DHE_EXPORT and 74.9% supporting DHE, but with the ten most common 1024-bit primes accounting for only 4.8% of servers.

If each of the top ten 1024-bit primes used by each protocol were compromised, this would affect approximately 1.7M SMTP, 276K IMAPS, and 245K POP3S servers. Using our downgrade attack of Section 6.3.3, an attacker with modest resources can hijack connections

to approximately 1.6M SMTP, 429K IMAPS, and 454K POP3S servers.

6.5. Recommendations

Our findings indicate that one of the key recommendations from security experts in response to the threat of mass surveillance—promotion of DHE-based TLS ciphersuites offering "perfect forward secrecy" over RSA-based ciphersuites—may have actually reduced security for many hosts. In this section, we present concrete recommendations to recover the expected security of Diffie-Hellman as it is used in mainstream Internet protocols.

Transition to elliptic curves. Transitioning to elliptic curve Diffie-Hellman (ECDH) key exchange with appropriate parameters avoids all known feasible cryptanalytic attacks. Current elliptic curve discrete log algorithms for strong curves do not gain as much of an advantage from precomputation. In addition, ECDH keys are shorter than in "mod p" Diffie-Hellman, and shared-secret computations are faster. Unfortunately, the most widely supported ECDH parameters, those specified by NIST, are now viewed with suspicion due to NSA influence on their design, despite no known or suspected weaknesses. These curves are undergoing scrutiny, and new curves, such as Curve25519, are being standardized by the IRTF for use in Internet protocols. We recommend transitioning to elliptic curves where possible; this is the most effective long-term solution to the vulnerabilities described in this paper.

Increase minimum key strengths. Server operators should disable DHE_EXPORT and configure DHE ciphersuites to use primes of 2048 bits or larger. Browsers and clients should raise the minimum accepted size for Diffie-Hellman groups to at least 1024 bits in order to avoid downgrade attacks when communicating with servers that still use smaller groups. Primes of less than 1024 bits should not be considered secure, even against an attacker with moderate resources.

Our analysis suggests that 1024-bit discrete log may be within reach for state-level actors. As such, 1024-bit DHE (and 1024-bit RSA) must be phased out in the near term. NIST has recommended such a transition since 2010 [50]. We recommend that clients raise the mini-

mum DHE group size to 2048 bits as soon as server configurations allow. Server operators should move to 2048-bit or larger groups to facilitate this transition. Precomputation for a 2048-bit non-trapdoored group is around 10^9 times harder than for a 1024-bit group, so 2048-bit Diffie-Hellman will remain secure barring a major algorithmic improvement.

Avoid fixed-prime 1024-bit groups. For implementations that must continue to use or support 1024-bit groups for compatibility reasons, generating fresh groups may help mitigate some of the damage caused by NFS-style precomputation for very common fixed groups. However, we note that it is possible to create trapdoored primes [153, 277] that are computationally difficult to detect. At minimum, clients should check that servers' parameters use safe primes or a verifiable generation process, such as that proposed in FIPS 186 [239]. Ideally, the process for generating and validating parameters in TLS should be standardized so as to thwart the risk of trapdoors.

Don't deliberately weaken crypto. Our downgrade attack on export-grade 512-bit Diffie-Hellman groups in TLS illustrates the fragility of cryptographic "front doors". Although the key sizes originally used in DHE_EXPORT were intended to be tractable only to NSA, two decades of algorithmic and computational improvements have significantly lowered the bar to attacks on such key sizes. Despite the eventual relaxation of crypto export restrictions and subsequent attempts to remove support for DHE_EXPORT, the technical debt induced by the additional complexity has left implementations vulnerable for decades. Like FREAK [61], our attacks warn of the long-term debilitating effects of deliberately weakening cryptography.

6.6. Disclosure and Response

We notified major client and server developers about the vulnerabilities discussed in this paper before we made our findings public. Prior to our work, Internet Explorer, Chrome, Firefox, and Opera all accepted 512-bit primes, whereas Safari allowed groups as small as 16 bits. As a result of our disclosures, Internet Explorer [222], Firefox, and Chrome are transitioning the minimum size of the DHE groups they accept to 1024 bits, and OpenSSL

and Safari are expected to follow suit. On the server side, we notified Apache, Oracle, IBM, Cisco, and various hosting providers. Akamai has removed all support for export ciphersuites. Many TLS developers plan to support a new extension that allows clients and servers to negotiate a few well-known groups of 2048-bits and higher and to gracefully reject weak ones [147].

6.7. Conclusion

Diffie-Hellman key exchange is a cornerstone of applied cryptography, but we find that, as used in practice, it is often less secure than widely believed. The problems stem from the fact that the number field sieve for discrete log allows an attacker to perform a single precomputation that depends only on the group, after which computing individual logs in that group has a far lower cost. Although this fact is well known to cryptographers, it apparently has not been widely understood by system builders. Likewise, many cryptographers did not appreciate that the security of a large fraction of Internet communication depends on Diffie-Hellman key exchanges that use a few small, widely shared groups.

A key lesson from this state of affairs is that cryptographers and creators of practical systems need to work together more effectively. System builders should take responsibility for being aware of applicable cryptanalytic attacks. Cryptographers, for their part, should involve themselves in how crypto is actually being applied, such as through engagement with standards efforts and software review. Bridging the perilous gap that separates these communities will be essential for keeping future systems secure.

CHAPTER 7 : DROWN attack and measurements

Abstract

We present DROWN, a novel cross-protocol attack that can decrypt passively collected TLS sessions from up-to-date clients by using a server supporting SSLv2 as a Bleichenbacher RSA padding oracle. We present two versions of the attack. The more general form exploits a combination of thus-far unnoticed protocol flaws in SSLv2 to develop a new and stronger variant of the Bleichenbacher attack. A typical scenario requires the attacker to observe 1,000 TLS handshakes, then initiate 40,000 SSLv2 connections and perform 2^{50} offline work to decrypt a 2048-bit RSA TLS ciphertext. (The victim client never initiates SSLv2 connections.) We implemented the attack and can decrypt a TLSv1.2 handshake using 2048-bit RSA in under 8 hours using Amazon EC2, at a cost of $440. Using Internet-wide scans, we find that 33% of all HTTPS servers and 22% of those with browser-trusted certificates are vulnerable to this protocol-level attack, due to widespread key and certificate reuse.

For an even cheaper attack, we apply our new techniques together with a newly discovered vulnerability in OpenSSL that was present in releases from 1998 to early 2015. Given an unpatched SSLv2 server to use as an oracle, we can decrypt a TLS ciphertext in one minute on a single CPU—fast enough to enable man-in-the-middle attacks against modern browsers. 26% of HTTPS servers are vulnerable to this attack.

We further observe that the QUIC protocol is vulnerable to a variant of our attack that allows an attacker to impersonate a server indefinitely after performing as few as 2^{25} SSLv2 connections and 2^{65} offline work.

We conclude that SSLv2 is not only weak, but actively harmful to the TLS ecosystem.

7.1. Introduction

TLS [106] is one of the main protocols responsible for transport security on the modern Internet. TLS and its precursor SSLv3 have been the target of a large number of cryp-

tographic attacks in the research community, both on popular implementations and the protocol itself [220]. Prominent recent examples include attacks on outdated or deliberately weakened encryption in RC4 [33], RSA [61], and Diffie-Hellman [29], different side channels including Lucky13 [32], BEAST [112], and POODLE [225], and several attacks on invalid TLS protocol flows [61, 65, 103].

Comparatively little attention has been paid to the SSLv2 protocol, likely because the known attacks are so devastating and the protocol has long been considered obsolete. Wagner and Schneier wrote in 1996 that their attacks on SSLv2 "will be irrelevant in the long term when servers stop accepting SSLv2 connections" [302]. Most modern TLS clients do not support SSLv2 at all. However, in Internet-wide scans we found that out of 36 million HTTPS servers, 6 million (17%) support SSLv2.

Bleichenbacher's padding oracle attack [73] is an adaptive chosen ciphertext attack against RSA PKCS#1 v1.5, the RSA padding standard used in TLS. This attack enables decryption of RSA-encrypted ciphertexts if a server distinguishes between correctly and incorrectly padded RSA plaintexts, and was termed the "million-message attack" upon its introduction in 1998 after the number of RSA decryption queries needed to deduce a plaintext. All widely-used modern SSL/TLS server implementations include countermeasures against Bleichenbacher attacks.

A Bleichenbacher attack on SSLv2. Our first result shows that the SSLv2 protocol is fatally vulnerable to a form of Bleichenbacher attack that enables decryption of RSA ciphertexts. We develop a novel application of the attack that allows us to use a server that supports SSLv2 as an efficient padding oracle. This attack is a protocol-level flaw in SSLv2 that results in a feasible attack for 40-bit export cipher strengths, and in fact abuses the universally implemented countermeasures against Bleichenbacher attacks to obtain a decryption oracle.

We also discovered multiple implementation flaws in commonly deployed OpenSSL versions

that allow an extremely efficient and much more dangerous instantiation of this attack.

Using SSLv2 to break TLS. Second, we present a novel *cross-protocol attack* that allows an attacker to break a passively collected RSA key exchange for any TLS server if the RSA keys are also used for SSLv2, possibly on a different server. We named our attack DROWN (*Decrypting RSA using Obsolete and Weakened eNcryption*).

In its *general* version, the attack exploits the protocol flaws in SSLv2, does not rely on any particular library implementation, and is feasible to carry out in practice for commonly supported export-grade ciphers. In order to decrypt one TLS session, the attacker must passively capture about 1,000 TLS sessions using RSA key exchange, make 40,000 SSLv2 connections to the victim server and perform 2^{50} symmetric encryption operations. We successfully carried out this attack using a heavily optimized GPU implementation and were able to decrypt a 2048-bit RSA ciphertext in less than 18 hours on a GPU cluster and less than 8 hours using the Amazon EC2 service.

We found that 11.5 million (33%) HTTPS servers are vulnerable to our attacks, because many HTTPS servers that do not directly offer SSLv2 share RSA keys with other services that do. Of servers offering HTTPS with browser-trusted certificates, 22% are vulnerable.

Our *special* version of the DROWN attack, which exploits a flaw in OpenSSL for a more efficient oracle, requires roughly the same number of captured TLS sessions, half as many connections to the victim server, and no large computations. The resulting attack can be completed on a single core on commodity hardware in less than a minute, without GPUs or distributed computing, and is limited primarily by how fast the server can complete handshakes. It is fast enough to perform man-in-the-middle attacks on live TLS sessions before the handshake times out, even allowing the attacker to target connections to servers that prefer non-RSA cipher suites and *downgrade* a modern TLS client to RSA key exchange. Our Internet-wide scans suggest that 79% of HTTPS servers that are vulnerable to the general attack, namely 26% of all HTTPS servers, are also vulnerable to real-time attacks

exploiting this dangerous implementation flaw.

Our results highlight the risk that continued support for SSLv2 imposes on the security of much more recent TLS versions. This is an instance of a more general phenomenon of insufficient domain separation, where older, vulnerable security standards can open the door to attacks on newer versions. We conclude that phasing out outdated and insecure standards should become a priority for standards designers and practitioners.

Responsible disclosure. The DROWN attack was assigned CVE-2016-0800. We disclosed our attacks to OpenSSL and worked with them to coordinate disclosure. The specific OpenSSL vulnerabilities we discovered have been assigned CVE-2015-3197 and CVE-2016-0703. In response to our disclosure, OpenSSL has made it impossible to configure a TLS server in such a way that it is vulnerable to DROWN. Microsoft had already disabled SSLv2 for all supported versions of IIS. We also disclosed the attack to the NSS developers, who have disabled SSLv2 on the last NSS tool that supported it, and have hastened their efforts to entirely remove support for the protocol from the NSS codebase. In response to our disclosure, Google will disable QUIC support for non-whitelisted servers, and make changes to the QUIC standard, as detailed in Section 7.7. We also notified IBM, Cisco, Amazon, the German CERT-Bund, and the Israeli CERT.

7.2. Background

In the following, $a||b$ denotes concatenation of strings a and b. $a[i]$ references the i-th byte in a. (N, e) denotes an RSA public key, where N has byte-length ℓ ($|N| = \ell$) and e is the public exponent. The corresponding secret exponent is $d = 1/e \bmod \phi(N)$.

7.2.1. PKCS#1 v1.5 encryption padding

Our attacks rely on the structure of RSA PKCS#1 v1.5 padding. Although there are newer versions of the PKCS standard, for example RSA PKCS#1 v2.0 which implements OAEP, SSL/TLS uses PKCS#1 v1.5. The basic task of the PKCS#1 v1.5 encryption padding scheme [186] is to randomize encryptions by prepending a random padding string PS to a message k (typically a symmetric session key) before applying RSA encryption:

1. The plaintext message is k. The encrypter generates a random byte string PS, where $|PS| \geq 8$, $|PS| = \ell - 3 - |k|$, and $\texttt{0x00} \notin \{PS[1], \ldots, PS[|PS|]\}$.

2. The encryption block is $m = 00||02||PS||00||k$.

3. The ciphertext is computed as $c = m^e \bmod N$.

To decrypt such a ciphertext, the decrypter first computes $m = c^d \bmod N$. Then it checks whether the decrypted message m is correctly formatted as a PKCS#1 v1.5-encoded message. We say that the ciphertext c and the decrypted message bytes $m[1]||m[2]||...||m[\ell]$ are PKCS#1 v1.5 conformant if:

$$\begin{aligned} m[1]||m[2] &= \texttt{0x00}||\texttt{0x02} \\ \texttt{0x00} &\notin \{m[3], \ldots, m[10]\} \end{aligned}$$

If this condition holds, the decrypter searches for the first value $i > 10$ such that $m[i] = 0x00$. Then, it extracts $k = m[i+1]|| \ldots ||m[\ell]$. Otherwise, the ciphertext is rejected.

In SSLv3 and TLS, RSA PKCS#1 v1.5 is used to encapsulate the premaster secret exchanged during the handshake [106]. Thus, k is interpreted as the premaster secret. In SSLv2, RSA PKCS#1 v1.5 is used for encapsulation of an equivalent key denoted the `master_key`.

7.2.2. SSL and TLS

The first incarnation of the TLS protocol was the SSL (Secure Socket Layer) protocol, which was designed by Netscape in the 90s. The first two versions of SSL were immediately found to be vulnerable to trivial attacks [294, 302] which were fixed in SSLv3 [129]. Later versions of the standard were renamed TLS, and share a similar structure to SSLv3. The current version of the protocol is TLSv1.2; TLSv1.3 is currently under development.

An SSL/TLS protocol flow consists of two phases: handshake and application data exchange. In the first phase, the communicating parties agree on cryptographic algorithms and establish shared keys. In the second phase, these keys are used to protect the confiden-

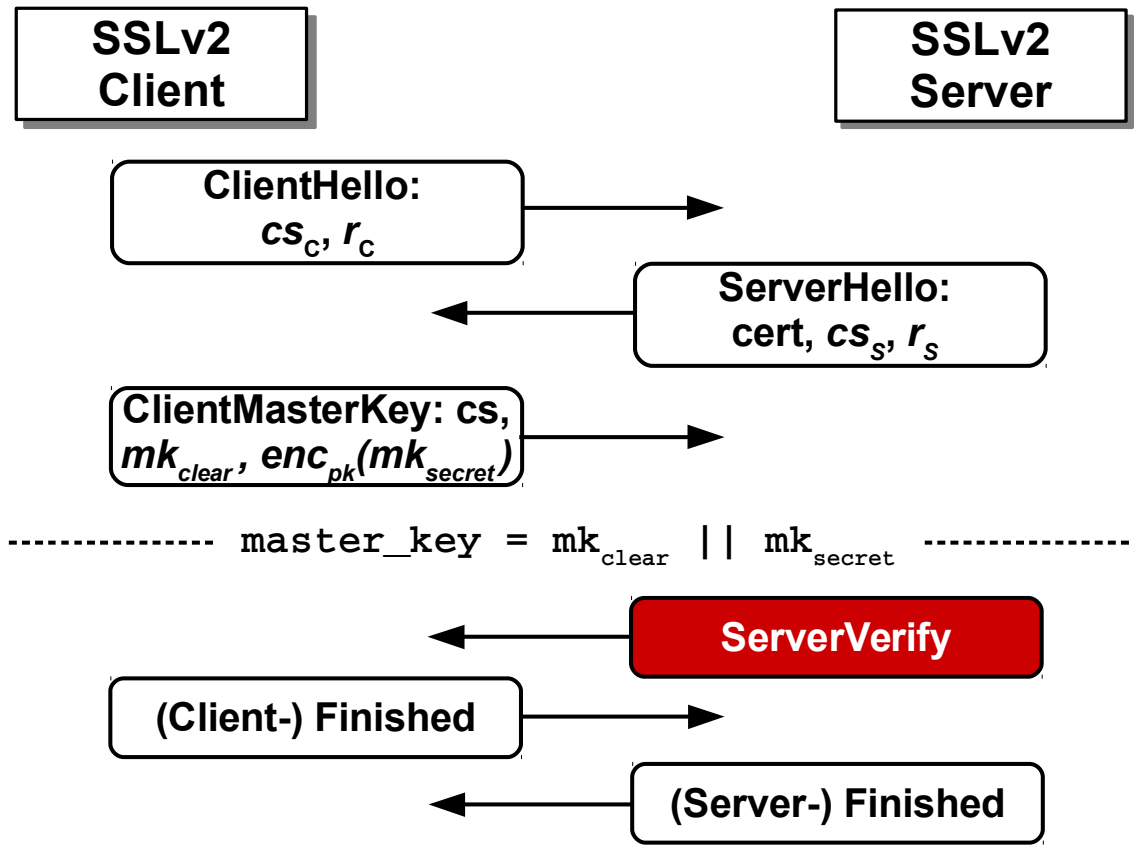


Figure 15: **SSLv2 handshake**—The server responds with a `ServerVerify` message directly after receiving an RSA-PKCS#1 v1.5 ciphertext contained in `ClientMasterKey`. This protocol feature enables our attack.

tiality and authenticity of the transmitted application data.

The handshake protocol was fundamentally redesigned in the SSLv3 version. This new handshake protocol was then used in later TLS versions up to TLSv1.2. In the following, we describe the RSA-based handshake protocols used in TLS and SSLv2, and highlight their differences.

The SSLv2 handshake protocol. The SSLv2 protocol description [167] is much less formally specified than modern RFCs. Figure 15 depicts an SSLv2 handshake. A client initiates an SSLv2 handshake by sending a `ClientHello` message, which includes a list of cipher suites cs_c supported by the client and a client nonce r_c, termed `challenge`. The server

responds with a `ServerHello` message, which contains a list of cipher suites cs_s supported by the server, the server certificate, and a server nonce r_s, termed `connection_ID`.

The client responds with a `ClientMasterKey` message, which specifies a cipher suite supported by both peers and key data used for constructing a `master_key`. In order to support *export* cipher suites with 40-bit security (e.g., `SSL_RC2_128_CBC_EXPORT40_WITH_MD5`), the key data is divided into two parts:

- mk_{clear}: A portion of the `master_key` sent in the `ClientMasterKey` message as plaintext (termed `clear_key_data` in the SSLv2 standard).
- mk_{secret}: A secret portion of the `master_key`, encrypted with RSA PKCS#1 v1.5 (termed `secret_key_data`).

The resulting `master_key` mk is constructed by concatenating these two keys: $mk = mk_{clear}||mk_{secret}$. For 40-bit export cipher suites, mk_{secret} is five bytes in length. For non-export cipher suites, the whole `master_key` is encrypted, and the length of mk_{clear} is zero.

The client and server can then compute session keys from the reconstructed `master_key` mk:

$$\texttt{server\_write\_key} = MD5(mk||\text{“0”}||r_c||r_s)$$
$$\texttt{client\_write\_key} = MD5(mk||\text{“1”}||r_c||r_s)$$

The server responds with a `ServerVerify` message consisting of the `challenge` r_c encrypted with the `server_write_key`. Both peers then exchange `Finished` messages in order to authenticate to each other.

Our attack exploits the fact the server always decrypts an RSA-PKCS#1 v1.5 ciphertext, computes the `server_write_key`, and *immediately* responds with a `ServerVerify` message. The SSLv2 standard implies this message ordering, but does not make it explicit. However, we observed this behavior in every implementation we examined. Our attack also takes

advantage of the fact that the encrypted mk_{secret} portion of the `master_key` can vary in length, and is only five bytes for export ciphers.

The TLS handshake protocol. In TLS [106] or SSLv3, the client initiates the handshake with a `ClientHello`, which contains a client random r_c and a list of supported cipher suites. The server chooses one of the cipher suites and responds with three messages, `ServerHello`, `Certificate`, and `ServerHelloDone`. These messages include the server's choice of cipher suite, server nonce r_s, and a server certificate with an RSA public key. The client then uses the public key to encrypt a newly generated 48-byte premaster secret *pms* and sends it to the server in a `ClientKeyExchange` message. The client and server then derive encryption and MAC keys from the premaster secret and the client and server random nonces. The details of this derivation are not important to our attack. The client then sends `ChangeCipherSpec` and `Finished` messages. The `Finished` message authenticates all previous handshake messages using the derived keys. The server responds with its own `ChangeCipherSpec` and `Finished` messages.

The two main details relevant to our attacks are:

- The premaster secret is always 48 bytes long, independent of the chosen cipher suite. This is also true for export cipher suites.
- After receiving the `ClientKeyExchange` message, the server waits for the `ClientFinished` message, in order to authenticate the client.

7.2.3. OpenSSL SSLv2 cipher suite selection bug

The SSLv2 protocol is supported in OpenSSL by default in all versions under 1.1.0. OpenSSL removed SSLv2 cipher suites from the default cipher string in 2010 between versions 0.9.8n and 1.0.0; the changelog discusses this as being equivalent to disabling support for SSLv2 by default [245]. Unfortunately, during our experiments we discovered that OpenSSL servers do not respect the cipher suites advertised in the `ServerHello` message. That is, the client can select an *arbitrary* cipher suite in the `ClientMasterKey` message and force the use of

export cipher suites even if they are explicitly disabled in the server configuration. The SSLv2 protocol itself was still enabled by default in the OpenSSL standalone server for the most recent OpenSSL versions prior to our disclosure.

We notified the OpenSSL team of this vulnerability, which was assigned CVE ID CVE-2015-3197. We have cooperated to develop a fix, which was included in OpenSSL releases 1.0.2f and 1.0.1r [245].

7.2.4. Bleichenbacher's attack

Bleichenbacher's attack is a padding oracle attack—it exploits the fact that RSA ciphertexts should decrypt to plaintexts compliant with the PKCS#1 v1.5 padding format. If an implementation receives an RSA ciphertext that decrypts to an invalid PKCS#1 v1.5 plaintext, it might naturally leak this information via an error message, by closing the connection, or by taking longer to process the error condition. This behavior can leak information about the plaintext that can be modeled as a cryptographic *oracle* for the decryption process. Bleichenbacher [73] demonstrated how such an oracle could be exploited to decrypt RSA ciphertexts.

Algorithm. In the simplest attack scenario, the attacker has a valid PKCS#1 v1.5 ciphertext c_0 that he wishes to decrypt to discover the message m_0. He has no access to the private RSA key, but instead has access to an oracle $\mathcal{O}$ that will decrypt a ciphertext c and inform the attacker whether the most significant two bytes match the required value for a correct PKCS#1 v1.5 padding:

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } m = c^d \bmod N \text{ starts with } \texttt{0x00 02} \\ 0 & \text{otherwise.} \end{cases}$$

If the oracle answers with `1`, the attacker knows that $2B \leq m \leq 3B - 1$, where $B = 2^{8(\ell-2)}$. The attacker can take advantage of RSA malleability to generate new candidate ciphertexts

for any s:

$$c = (c_0 \cdot s^e) \bmod N = (m_0 \cdot s)^e \bmod N$$

The attacker queries the oracle with c. If the oracle responds with 0, the attacker increments s and repeats the previous step. Otherwise, the attacker learns that for some r, $2B \leq m_0 s - rN < 3B$. This allows the attacker to reduce the range of possible solutions to

$$\frac{2B + rN}{s} \leq m_0 < \frac{3B + rN}{s}$$

The attacker proceeds by refining guesses for s and r values and successively decreasing the size of the interval containing m_0. At some point the interval will contain a single valid value, m_0. Bleichenbacher's original paper describes this process in further detail [73].

Countermeasures. In order to protect against this attack, the decrypter must not leak any information about the PKCS#1 v1.5 validity of the ciphertext. Since the ciphertext itself does not decrypt to a valid message, the decrypter needs to generate a fake plaintext and continue with the protocol using this decoy. The attacker should not be able to distinguish the resulting computation from a correctly decrypted ciphertext.

In the case of SSL/TLS, the server generates a random premaster secret and finishes the handshake with this random premaster secret if the decrypted ciphertext is invalid. The client will not possess the session key to send a valid `ClientFinished` message and the connection will terminate.

7.3. Breaking TLS with SSLv2

In this section, we describe our cross-protocol DROWN attack that uses an SSLv2 server as an oracle to efficiently decrypt TLS connections. We first describe our techniques using a generic SSLv2 oracle. In Section 7.4, we show how a protocol flaw in SSLv2 can be used to construct such an oracle, and describe our general DROWN attack. In Section 7.5, we show how an implementation flaw in common versions of OpenSSL leads to a very powerful oracle, and describe our efficient special DROWN attack.

7.3.1. Attack scenario

We consider a server that accepts TLS connections from clients. The connections are established using a secure, state-of-the-art TLS version (1.0–1.2) and a `TLS_RSA` cipher suite where the private key is not known to the attacker.

Server RSA key exposed via SSLv2. The same RSA public key as the TLS connections is also used for SSLv2. For simplicity, our presentation will refer to the servers accepting TLS and SSLv2 connections as the same entity.

The attacker's position in the network. Our attacker is able to passively eavesdrop on traffic between the client and server and record RSA-based TLS traffic, but does not perform any active man-in-the-middle interference.

The attacker can expect to decrypt one out of 1,000 intercepted TLS connections in our attack for typical parameters. This is a devastating threat in many scenarios. For example, a decrypted TLS connection might reveal a client's HTTP cookie or plaintext password, and an attacker would only need to successfully decrypt a single ciphertext to compromise the client's account.

In order to collect 1,000 TLS connections, the attacker might simply wait patiently until sufficiently many connections are recorded. If the attacker's intended victim is the *server*, rather than a specific client, observing this many connections from many clients might take only a short time for an attacker who is located at a company firewall or who could perform a DNS spoofing or BGP hijacking attack to redirect traffic transparently through themselves. If the attacker's intended victim is a *particular client*, this is still feasible in many cases. As an example, the Mozilla Thunderbird email client will check for new email messages every ten minutes by default. A targeted user will make 1,000 connections after leaving the application running for a week. A less patient attacker could embed or inject malicious JavaScript on an otherwise innocuous web site to cause the client to connect repeatedly to the victim server in a short time frame, as in the BEAST attack [112]. Normally such

connections would use TLS session resumption instead of completing a fresh handshake on each time, but if an attacker can trigger an error, the next connection will be negotiated with a fresh handshake.

7.3.2. A generic SSLv2 oracle

Our attacks make use of a padding oracle that can be queried on a ciphertext and leaks information about decrypted plaintext; this abstractly models the information gained from an SSLv2 server's behavior. Our SSLv2 oracles reveal many bytes of plaintext, resulting in an efficient attack.

Our cryptographic oracle $\mathcal{O}$ has the following functionality: $\mathcal{O}$ decrypts an RSA ciphertext c and responds with ciphertext validity based on the structure of the decrypted message m. The ciphertext is valid only if m starts with `0x00 02` followed by non-null padding bytes, a delimiter byte `0x00`, and a `master_key` mk_{secret} of correct byte length k. In the following, we denote such a ciphertext to be *SSLv2 conformant*.

All of the SSLv2 padding oracles we instantiate give the attacker similar information about a PKCS#1 v1.5 conformant SSLv2 ciphertext:

$$\mathcal{O}(c) = \begin{cases} mk_{secret} & \text{if } c^d \bmod N = 00||02||PS||00||mk_{secret} \\ 0 & \text{otherwise.} \end{cases}$$

That is, the oracle $\mathcal{O}(c)$ will return the decrypted message mk_{secret} if it is queried on a PKCS#1 v1.5 conformant SSLv2 ciphertext c corresponding to a correctly PKCS#1 v1.5 padded encryption of mk_{secret}. The attacker then learns $k+3$ bytes of information about $m = c^d \bmod N$: the first two bytes are $00||02$, and the last $k+1$ bytes are $00||mk_{secret}$. The length k of mk_{secret} varies based on the cipher suite used in the instantiation of the oracle. For export-grade cipher suites such as `SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5`, k will be 5 bytes, so the attacker learns 8 bytes of information about m. For `SSL_DES_192_EDE3_CBC_WITH_MD5`, k is 24 bytes and the attacker learns 27 bytes of plaintext.

7.3.3. DROWN attack template

Our attacker will use an SSLv2 oracle $\mathcal{O}$ to decrypt a TLS `ClientKeyExchange`. The behavior of $\mathcal{O}$ poses two problems for the attacker. First, a TLS ciphertext transmitted in a TLS key exchange decrypts to a 48-byte premaster secret. But since no SSLv2 cipher suites have 48-byte key strengths, this means that a valid TLS ciphertext is invalid to our oracle $\mathcal{O}$. In order to apply Bleichenbacher's attack, the attacker needs to transform the TLS ciphertext into a valid SSLv2 key exchange message. Second, $\mathcal{O}$ is very restrictive, since it strictly checks the length of the unpadded message. According to Bardou et al. [48], using such an oracle for Bleichenbacher's attack would require 12 million oracle queries.[1]

Our attacker overcomes these problems by following this generic attack flow:

0. The attacker collects many encrypted TLS RSA key exchange messages.

1. He then attempts to convert the intercepted TLS ciphertexts containing a 48-byte premaster secret to valid RSA PKCS#1 v1.5 encoded ciphertexts containing messages of length appropriate to the SSLv2 oracle $\mathcal{O}$. We accomplish this by taking advantage of RSA ciphertext malleability and a technique of Bardou et al. [48].

2. Once the attacker has obtained a valid SSLv2 RSA ciphertext, he can continue with a modified version of Bleichenbacher's attack, and decrypt the message after many more oracle queries.

3. The attacker can then transform the decrypted plaintext back into the original plaintext, which is one of the collected TLS handshakes.

We describe the algorithmic improvements we use to make each of these steps efficient below.

Finding an SSLv2 conformant ciphertext. The first step for the attacker is to transform the original TLS `ClientKeyExchange` message c_0 from a TLS conformant ciphertext

[1] See Table 1 in [48]. The oracle is denoted with the term `FFF`.

into an SSLv2 conformant ciphertext. A trivial approach would be to generate multipliers $s_i \in \{s_1, s_2, \ldots\}$, and compute ciphertexts $c_i = (c_0 {s_i}^e) \bmod N$, until one gets accepted by $\mathcal{O}$. However, the number of generated ciphertexts would be high, because $\mathcal{O}$ is very restrictive; for 2048-bit RSA keys and an oracle returning a 5-byte k the probability that a random ciphertext becomes SSLv2 conformant is $P_{rnd} \approx (1/256)^3 * (255/256)^{249} \approx 2^{-25}$.

Instead, we rely on the concept of *trimmers*, which were introduced by Bardou et al. [48]. Assume that the message $m_0 = {c_0}^d \bmod N$ is divisible by a small number t. In that case, $m_0 \cdot t^{-1} \bmod N$ simply equals the natural number m_0/t. If we choose $u \approx t$, and multiply the original message with a fraction u/t, the resulting number will lie near the original message: $m_0 \approx m_0/t \cdot u$. We shall refer to such fractions as "small" fractions.

This method allows us to generate new SSLv2 conformant messages with a much higher probability. Let c_0 be an intercepted TLS conformant RSA ciphertext, and let $m_0 = c_0^d \bmod N$ be its corresponding plaintext. We select a multiplier $s = u/t \bmod N = ut^{-1} \bmod N$ where u and t are coprime, compute the value $c_1 = c_0 s^e \bmod N$, and query $\mathcal{O}(c_1)$. We will receive a response if $m_1 = m_0 \cdot u/t$ is SSLv2 conformant.

As an example, let us assume a 2048-bit RSA ciphertext with $k = 5$, and consider the fraction $u = 7, t = 8$. The probability that a random ciphertext c_0 will be SSLv2 conformant is 1/7,774, so we expect to make 7,774 oracle queries before discovering a ciphertext c_0 for which $c_0 u/t$ is SSLv2 conformant, much better than a randomly selected multiplier. Appendix 7.B.1 gives more details on computing these probabilities.

Shifting known plaintext bytes. Once we have obtained an SSLv2 conformant ciphertext c_1, we have also learned from our oracle information about the $k + 1$ least significant bytes (mk_{secret} together with the delimiter byte `0x00`) and two most significant `0x00 02` bytes of the SSLv2 conformant message m_1. We would like to *rotate* these known bits around to the right, so that we have a large block of contiguous known most significant bytes of plaintext. In this section, we show that this can be accomplished by

multiplying by some shift $2^{-r} \bmod N$. In other words, given an SSLv2 conformant ciphertext $c_1 = m_1^e \bmod N$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \bmod N$ where $m_2 = s \cdot m_1 \cdot 2^{-r} \bmod N$ and we know several most significant bytes of m_2.

Let $R = 2^{8(k+1)}$ and $B = 2^{8(\ell-2)}$. Abusing notation slightly, let the integer $m_1 = 2 \cdot B + PS \cdot R + mk_{secret}$ be the plaintext satisfying $m_1^e = c_1 \bmod N$. At this stage, the k-byte integer mk_{secret} is known and the $\ell - k - 3$-byte integer PS is not.

Let $\tilde{m_1} = 2 \cdot B + mk_{secret}$ be the known components of m_1, so $m_1 = \tilde{m_1} + PS \cdot R$. We can use this to compute a new plaintext for which we know many most significant bytes. Consider the value

$$m_1 \cdot R^{-1} \bmod N = \tilde{m_1} \cdot R^{-1} + PS \bmod N.$$

The value of PS is unknown, but we know that it consists of $\ell - k - 3$ bytes. This means that the known value $\tilde{m_1} \cdot R^{-1}$ shares most of its $k+3$ most significant bytes with $m_1 \cdot R^{-1}$.

Furthermore, we can iterate this process by finding a new multiplier s such that $m_2 = s \cdot m_1 \cdot R^{-1} \bmod N$ is also SSLv2 conformant. A randomly chosen $s < 2^{30}$ will work with probability $2^{-25.4}$. We can take advantage of the bytes we have already learned about m_1 to efficiently compute such an s with only 678 oracle queries in expectation for a 2048-bit RSA modulus. Appendix 7.B.3 gives more details.

Adapted Bleichenbacher iteration. It is feasible for all of our oracles to use the previous technique to entirely recover a plaintext message. However, for our SSLv2 protocol oracle it is cheaper to continue using Bleichenbacher's original attack, once we have used the above techniques to obtain a SSLv2 conformant message m_3 and an integer s_3 such that $m_3 \cdot s_3$ is SSLv2 conformant. At this point, we can apply the original algorithm proposed by Bleichenbacher as described in Section 7.2.4, with minimal modifications.

Each step obtains a message that starts with the required `0x00 02` bytes after two queries in

expectation. Since we know the value of the $k+1$ least significant bytes after multiplying by any integer, we can query the oracle only on multipliers that cause the $(k+1)$st least significant byte to be zero. However, we cannot ensure that the padding string is entirely nonzero; for a 2048-bit modulus this will hold with probability 0.37.

For a 2048-bit modulus, the total expected number of queries when using this technique to fully decrypt the plaintext is $2048 * 2/0.37 \approx 11,000$.

7.4. General DROWN

In this section, we describe how any correct SSLv2 implementation that accepts export-grade cipher suites can be used as a padding oracle. We then show how to adapt the techniques described in Section 7.3.3 to decrypt TLS RSA ciphertexts.

7.4.1. The SSLv2 export padding oracle

SSLv2 is vulnerable to a direct message side channel vulnerability exposing a Bleichenbacher oracle to the attacker. The vulnerability follows from three properties of SSLv2. First, the server immediately responds with a `ServerVerify` message after receiving the `ClientMasterKey` message, which includes the RSA ciphertext, without waiting for the `ClientFinished` message that proves the client knows the RSA plaintext. Second, when choosing 40-bit export RC2 or RC4 as the symmetric cipher, only 5 bytes of the `master_key` (mk_{secret}) are sent encrypted using RSA, and the remaining 11 bytes are sent in cleartext. Third, a server implementation that correctly implements the anti-Bleichenbacher countermeasure and receives an RSA key exchange message with invalid padding will generate a random premaster secret and carry out the rest of the TLS handshake using this randomly generated key material.

This allows an attacker to deduce the validity of RSA ciphertexts in the following manner:

1. The attacker sends a `ClientMasterKey` message, which contains an RSA ciphertext c_0 and any sequence of 11 bytes as the clear portion of the `master_key`, mk_{clear}. The server responds with a `ServerVerify` message, which contains the `challenge` encrypted using the `server_write_key`.

2. The attacker performs an *exhaustive search* over the possible values of the 5 bytes of the `master_key` mk_{secret}. He then computes the corresponding `server_write_key` and checks whether the `ServerVerify` message decrypts to the `challenge`. One value should pass this check; let this value be termed mk_0. Recall that if the RSA plaintext was valid, mk_0 is the unpadded data in the RSA plaintext. Otherwise, mk_0 is a randomly generated sequence of 5 bytes.

3. The attacker re-connects to the server with the same RSA ciphertext c_0. The server responds with another `ServerVerify` message that contains the current `challenge` encrypted using the current `server_write_key`. If the decrypted RSA ciphertext was valid, the attacker can directly decrypt a correct `challenge` value from the `ServerVerify` message by using the `master_key` mk_0. Otherwise, if the `ServerVerify` message does not correctly decrypt to the `challenge`, the RSA ciphertext was invalid, and the attacker knows the mk_0 value was generated at random.

Thus we can instantiate an oracle $\mathcal{O}_{\mathsf{SSLv2\text{-}export}}$ using the procedure above; each oracle query requires two server connections and 2^{40} decryption attempts in the simplest case. For each oracle call $\mathcal{O}_{\mathsf{SSLv2\text{-}export}}(c)$, the attacker learns whether c is valid, and if so, learns the two most significant bytes `0x00 02`, the sixth least significant `0x00` delimiter byte, and the value of the 5 least significant bytes of the plaintext m.

If the server does not support 40-bit export ciphers, the attack can also be mounted in feasible computation time by choosing DES as the symmetric cipher. Choosing DES means the exhaustive search is now done over a key space of 56 bits, thus increasing the cost of the attack by a factor of 2^{16}, but does not fundamentally change anything except the increased cost.

7.4.2. TLS decryption attack

In this section, we describe how the oracle described in Section 7.4.1 can be used to carry out a feasible attack to decrypt passively collected TLS ciphertexts.

Attack scenario. As described in Section 7.3.1, we consider a server that accepts TLS connections from clients using an RSA public key that is exposed via SSLv2, and an attacker who is able to passively observe these connections.

We also assume the server supports export cipher suites for SSLv2. This can happen for two reasons. First, the same servers that fail to follow best practices in disabling SSLv2 [294] may also fail to follow best practices by supporting export cipher suites. Alternatively, the servers might be running a version of OpenSSL prior to January 2016, in which case they are vulnerable to the OpenSSL cipher suite selection bug described in Section 7.2.3, and an attacker may negotiate a cipher suite of his choice independent of the server configuration.

We assume the server implements the recommended countermeasure against Bleichenbacher's attack in all protocol versions, including SSLv2. If the decrypted RSA ciphertext has invalid padding, the server generates a random premaster secret or `master_key` and continues the handshake with this random string. We assume this countermeasure is implemented correctly and the server is neither vulnerable to timing nor flush-and-reload side-channel attacks [221, 317].

The attacker needs access to computing power sufficient to perform a 2^{50} time attack, mostly brute forcing symmetric key encryption. After our optimizations, this can be done with a one-time investment of a few thousand dollars of GPUs, or in a few hours for a few hundred dollars in the cloud. Our cost estimates are described in Section 7.4.3.

Constructing the attack. The attacker can exploit the SSLv2 vulnerability as illustrated in Figure 16, following the generic attack outline described in Section 7.3.3 and has several distinct phases:

0. He passively collects 1,000 TLS handshakes from connections using RSA key exchange.

1. The attacker then attempts to convert the intercepted TLS ciphertexts containing a 48-byte premaster secret to valid RSA PKCS#1 v1.5 encoded ciphertexts containing

five-byte messages using the fractional trimmers described in Section 7.3.3, and querying $\mathcal{O}_{\mathsf{SSLv2\text{-}export}}$. The attacker sends the modified ciphertexts to the server using fresh SSLv2 connections with weak symmetric ciphers and uses the `ServerVerify` messages to deduce ciphertext validity as described in the previous section. For each queried RSA ciphertext, the attacker must perform a brute force attack on the weak symmetric cipher. The attacker expects to obtain a valid SSLv2 ciphertext after roughly 10,000 oracle queries, or 20,000 connections to the server.

2. Once the attacker has obtained a valid SSLv2 RSA ciphertext m_1, he uses the shifting technique explained in Section 7.3.3 to find an integer s_1 such that $m_2 = m_1 \cdot 2^{-40} \cdot s_1$ is also SSLv2 conformant. Appendix 7.B.4 contains more details on this step.

3. The attacker then applies the shifting technique again to find another integer s_2 such that $m_3 = m_2 \cdot 2^{-40} \cdot s_2$ is also SSLv2 conformant.

4. He then searches for yet another integer s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant.

5. Finally, the attacker can continue with our adapted Bleichenbacher iteration technique described in Section 7.3.3, and decrypts the message after an expected 10,000 additional oracle queries, or 20,000 connections to the server.

6. The attacker can then transform the decrypted plaintext back into the original plaintext, which is one of the 1,000 intercepted TLS handshakes.

Bleichenbacher's original algorithm requires a conformant message m_0, and a multiplier s_1 such that $m_1 = m_0 \cdot s_1$ is also conformant. Naïvely, it would appear we can apply the same algorithm here, after completing Phase 1. However, the original algorithm expects s_1 to be of size about 2^{24}. This is not the case when we use fractions for s_1, as the integer $s_1 = ut^{-1} \bmod N$ will be the same size as N.

Therefore, our approach is to find a conformant message for which we know the 5 most significant bytes; this will happen after multiple rotations and this message will be m_3.

Optimizing for	Ciphertexts	$\|F\|$	SSLv2 connections	Offline work
offline work	12,743	1	50,421	$2^{49.64}$
offline work	1,055	10	46,042	$2^{50.63}$
compromise	4,036	2	41,081	$2^{49.98}$
online work	2,321	3	38,866	$2^{51.99}$
online work	906	8	39,437	$2^{52.25}$

Table 34: **2048-bit Bleichenbacher attack complexity**—The cost to decrypt one ciphertext can be adjusted by choosing the set of fractions F the attacker applies to each of the passively collected ciphertexts in the first step of the attack. This choice affects several parameters: the number of these collected ciphertexts, the number of connections the attacker makes to the SSLv2 server, and the number of offline decryption operations.

Key size	Phase 1	Phases 2–5	Total queries	Offline work
1024	4,129	4,132	8,261	$2^{50.01}$
2048	6,919	12,468	19,387	$2^{50.76}$
4096	18,286	62,185	80,471	$2^{52.16}$

Table 35: **Oracle queries required by our attack**—In Phase 1, the attacker queries the oracle until an SSLv2 conformant ciphertext is found. In Phases 2–5, the attacker decrypts this ciphertext using leaked plaintext. These numbers minimize total queries. In our attack, an oracle query represents two server connections.

After finding such a message, finding s_3 such that $m_4 = m_3 \cdot s_3$ is also conformant becomes trivial. From there, we can finally apply the adapted Bleichenbacher iteration technique as described in Appendix 7.B.5.

Attack performance. The attacker wishes to minimize three major costs in the attack: the number of recorded ciphertexts from the victim client, the number of connections to the victim server, and the number of symmetric keys to be brute forced. The requirements for each of these elements are governed by the set of fractions to be multiplied with each RSA ciphertext in the first phase, as described in Section 7.3.3.

Table 34 highlights a few choices for F and the resutling performance metrics for 2048-bit RSA keys. Appendix 7.B.6 provides more details on the derivation of these numbers and other possible optimization choices. Table 35 gives the expected number of Bleichenbacher

queries for different RSA key sizes, when minimizing total oracle queries.

7.4.3. Implementing general DROWN with GPUs

The most computationally expensive part of our general DROWN attack is breaking the 40-bit symmetric key, so we developed a highly optimized GPU implementation of this brute force attack. Our first naïve GPU implementation performed around 26MH/s, where MH measures the calculation of an MD5 hash and the RC2 decryption. Our optimized implementation gave a final speed of 515MH/s, a speedup factor of 19.8.

We obtained our improvements through a number of optimizations. Our original implementation ran into a communication bottleneck in the PCI-E bus in transmitting candidate keys from CPU to GPU, so we removed this bottleneck by generating key candidates on the GPU itself. We optimized memory management, including storing candidate keys and the RC2 permutation table in constant memory, which is almost as fast as a register, instead of slow global memory. We optimized the cryptographic checks themselves by rewriting the RC2 implementation to use 32-bit instructions, removing unnecessary RC2 keysize checks, dropping unused ADD instructions during MD5, and manually shifting input bytes into the MD5 input registers to avoid loop branches. We describe these optimizations in further detail in Appendix 7.C.

We experimentally evaluated our optimized implementation on a local cluster and in the cloud. We used it to execute a full attack of $2^{49.6}$ tested keys on each platform. The required number of keys to test during the attack is a random variable, distributed geometrically, with an expectation that ranges between $2^{49.6}$ and $2^{52.5}$ depending on the choice of optimization parameters. We treat a full attack as requiring $2^{49.6}$ tested keys overall.

Hashcat. Hashcat[4] is an open source optimized password-recovery tool. The Hashcat developers allowed us to use their GPU servers for our attack evaluation. The servers contain a total of 40 GPUs: 32 Nvidia GTX 980 cards, and 8 AMD R9 290X cards. The value of this equipment is roughly $18,040. Our full attack took less than 18 hours to complete on the Hashcat servers, with the longest single instance taking 17h9m.

Amazon EC2. We also ran our optimized GPU code on the Amazon Elastic Compute Cloud (EC2) [37] service. We used a cluster composed of 200 variable-price "spot" instances: 150 `g2.2xlarge` instances, each of which contains one high-performance NVIDIA GPU with 1,536 CUDA cores and 50 `g2.8xlarge` instances, each containing four of these GPUs. When we ran our experiments in January 2016, the average spot rates we paid were $0.09/hr and $0.83/hr respectively. Our full attack finished in under 8 hours including startup and shutdown for a cost of $440. See Appendix 7.D for more details.

7.5. Special DROWN

We discovered a vulnerability in recent (but not current) versions of the OpenSSL SSLv2 handshake code that creates a powerful Bleichenbacher oracle, and drastically reduces the amount of computation required to implement our attack. The vulnerability, which has been designated CVE-2016-0703, was present in the OpenSSL codebase from at least the start of the repository, in 1998, until it was unknowingly fixed on March 4, 2015 by a patch [187] designed to correct an unrelated problem [25]. By adapting DROWN to exploit this special case, we can cut the number of connections required by more than 50% and reduce the computational work to a negligible amount.

7.5.1. The OpenSSL "extra clear" oracle

Prior to the fix, OpenSSL servers improperly allowed the `ClientMasterKey` message to contain `clear_key_data` bytes for *non-export* ciphers. When such bytes are present, the server substitutes them for bytes from the encrypted key. For example, consider the case that the client chooses a 128-bit cipher and sends a 16-byte encrypted key $k[1], k[2], \ldots, k[16]$ but, contrary to the protocol specification, includes 4 null bytes of `clear_key_data`. Vulnerable OpenSSL versions will construct the following `master_key`:

$$[00\ 00\ 00\ 00\ k[1]\ k[2]\ k[3]\ k[4]\ \ldots\ k[9]\ k[10]\ k[11]\ k[12]]$$

This enables a straightforward key-recovery attack against such versions. An attacker that has intercepted an SSLv2 connection takes the RSA ciphertext of the encrypted key and replays it in non-export handshakes to the server with varying lengths of `clear_key_data`. For a 16-byte encrypted key, the attacker starts with 15 bytes of clear key, causing the

server to use the `master_key`:

$$[00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ k[1]]$$

The attacker can brute force the first byte of the encrypted key by finding the matching `ServerVerify` message among 256 possibilities. Knowing the first byte, the attacker makes another connection with the same RSA ciphertext but 14 bytes of clear key, resulting in the `master_key`:

$$[00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ k[1]\ k[2]]$$

Since the attacker already knows $k[1]$, he can easily brute force the second byte. With only 15 probe connections and an expected $15 \cdot 128 = 1,920$ trial encryptions, the attacker learns the entire `master_key` for the recorded session.

This session key-recovery attack can be directly converted to a Bleichenbacher oracle. Given a candidate ciphertext and symmetric key length k, the attacker sends the ciphertext with k known bytes of `clear_key_data`. The oracle decision is simple:

- If the ciphertext is valid, the `ServerVerify` message will reflect a `master_key` consisting of those k known bytes.
- If the ciphertext is invalid, the `master_key` will be replaced with k random bytes (by following the countermeasure against the Bleichenbacher attack), resulting in a different `ServerVerify` message.

This oracle decision requires one connection to the server and one `ServerVerify` computation. After the attacker has found a valid ciphertext corresponding to a k-byte encrypted key, they can recover the k plaintext bytes by repeating the key recovery attack from above. Thus our oracle $\mathcal{O}_{\mathsf{SSLv2\text{-}extra\text{-}clear}}(c)$ requires one connection to determine whether c is valid, and thus the two most significant bytes `0x00 02` of the plaintext m. After k connections, the attacker can additionally learn the k least significant bytes of m. We model this as a single oracle call, but the number of server connections will vary depending on the response.

Protocol	Port	*All Certificates*			*Trusted certificates*		
		SSL/TLS	**SSLv2 support**	**Vulnerable key**	**SSL/TLS**	**SSLv2 support**	**Vulnerable key**
SMTP	25	3,357 K	936 K (28%)	1,666 K (50%)	1,083 K	190 K (18%)	686 K (63%)
POP3	110	4,193 K	404 K (10%)	1,764 K (42%)	1,787 K	230 K (13%)	1,031 K (58%)
IMAP	143	4,202 K	473 K (11%)	1,759 K (42%)	1,781 K	223 K (13%)	1,022 K (57%)
HTTPS	443	34,727 K	5,975 K (17%)	11,444 K (33%)	17,490 K	1,749 K (10%)	3,931 K (22%)
SMTPS	465	3,596 K	291 K (8%)	1,439 K (40%)	1,641 K	40 K (2%)	949 K (58%)
SMTP	587	3,507 K	423 K (12%)	1,464 K (42%)	1,657 K	133 K (8%)	986 K (59%)
IMAPS	993	4,315 K	853 K (20%)	1,835 K (43%)	1,909 K	260 K (14%)	1,119 K (59%)
POP3S	995	4,322 K	884 K (20%)	1,919 K (44%)	1,974 K	304 K (15%)	1,191 K (60%)
(Alexa 1M)	443	611 K	82 K (13%)	152 K (25%)	456 K	38 K (8%)	109 K (24%)

Table 36: **Hosts vulnerable to general DROWN**—We performed Internet-wide scans to measure the number of hosts supporting SSLv2 on several different protocols. A host is vulnerable to DROWN if its public key is exposed anywhere via SSLv2. Overall vulnerability to DROWN is much larger than support for SSLv2 due to widespread reuse of keys.

7.5.2. TLS decryption with special DROWN

Using our oracle $\mathcal{O}_{\mathsf{SSLv2\text{-}extra\text{-}clear}}$, we can construct an extremely efficient version of our TLS decryption attack. The OpenSSL extra clear oracle provides three significant advantages over our export oracle $\mathcal{O}_{\mathsf{SSLv2\text{-}export}}$: (1) It no longer requires an export cipher suite, and, in fact, we gain efficiency by exploiting regular SSLv2 ciphers; (2) It requires only one handshake per oracle query; and (3) Computation is reduced to one `ServerVerify` decryption per oracle query, versus 2^{40}.

Attack scenario. As before, we consider a server that accepts TLS connections, and a client that negotiates a secure, state-of-the-art TLS version with a `TLS_RSA` cipher suite. The same RSA key pair used for TLS is also used on a server that is running a vulnerable version of OpenSSL.

Constructing the attack. The attacker can exploit the OpenSSL extra clear vulnerability to efficiently decrypt a TLS ciphertext as follows. We will use the cipher suite `SSL_DES_192_EDE3_CBC_WITH_MD5` as the cipher suite, allowing the attacker to recover 24 bytes of key at a time from the oracle. We first present a straightforward adaptation of the general DROWN attack to the extra clear oracle, before later applying a few additional optimizations made possible by this new oracle.

0. The attacker intercepts several hundred TLS handshakes using RSA key exchange.

1. The attacker uses the fractional trimmers as described in Section 7.3.3 to convert the TLS ciphertexts into an SSLv2 conformant ciphertext c_0.

2. Once the attacker has obtained a valid SSLv2 ciphertext c_1, he repeatedly uses the shifting technique described in Section 7.3.3 to rotate the message by 25 bytes each iteration, learning 27 bytes with each shift. After several iterations, he has learned the entire plaintext.

3. The attacker then transforms the decrypted SSLv2 plaintext into the decrypted TLS plaintext.

Attack costs. Using 40 fractional trimmers, this more efficient oracle attack allows the attacker to recover one in 260 TLS session keys using only about 17,000 connections to the server. The computation cost is so low that we can complete the full attack on a single workstation in under one minute. Appendix 7.B.7 gives more details.

Mounting the attack using the optimized version of Special DROWN described in Appendix 7.B.7 allows the attacker to target one of 100 connections, at the expense of increasing the number of queries to 27,000.

7.5.3. MITM attack against TLS

Special DROWN is fast enough that it can decrypt a TLS premaster secret *online*, during a connection handshake. A man-in-the-middle attacker can use it to compromise connections between modern browsers and TLS servers—even those configured to prefer non-RSA cipher suites.

Attack scenario. The MITM attacker impersonates the server and sends a `ServerHello` message that selects a cipher suite with RSA as the key-exchange method. Then, the attacker uses special DROWN to decrypt the premaster secret. The main difficulty is completing the decryption and producing a valid `ServerFinished` message before the client's connection times out. Most browsers will allow the handshake to last up to one minute [29].

		Any certificate			*Trusted certificates*		
Protocol	**Port**	**SSL/TLS**	**Special DROWN oracles**	**Vulnerable key**	**SSL/TLS**	**Vulnerable key**	**Vulnerable name**
SMTP	25	3,357 K	855 K (25%)	896 K (27%)	1,083 K	305 K (28%)	398 K (37%)
POP3	110	4,193 K	397 K (9%)	946 K (23%)	1,787 K	485 K (27%)	674 K (38%)
IMAP	143	4,202 K	457 K (11%)	969 K (23%)	1,781 K	498 K (30%)	690 K (39%)
HTTPS	443	34,727 K	4,029 K (12%)	9,089 K (26%)	17,490 K	2,523 K (14%)	3,793 K (22%)
SMTPS	465	3,596 K	334 K (9%)	765 K (21%)	1,641 K	430 K (26%)	630 K (38%)
SMTP	587	3,507 K	345 K (10%)	792 K (23%)	1,657 K	482 K (29%)	667 K (40%)
IMAPS	993	4,315 K	892 K (21%)	1,073 K (25%)	1,909 K	602 K (32%)	792 K (42%)
POP3S	995	4,322 K	897 K (21%)	1,108 K (26%)	1,974 K	641 K (32%)	835 K (42%)
(Alexa 1M)	443	611 K	22 K (4%)	52 K (9%)	456 K (100%)	33 K (7%)	85 K (19%)

Table 37: **Hosts vulnerable to special DROWN**—A server is vulnerable to special DROWN if its key is exposed by a host with the CVE-2016-0703 bug. Since the attack is fast enough to enable man-in-the-middle attacks, a server is also vulnerable (to impersonation) if any name in its certificate is found in any trusted certificate with an exposed key.

Using the fully optimized version of special DROWN, the attack still requires intercepting an average of 100 ciphertexts, only one of which will be decrypted, probabilistically. The simplest way for the attacker to facilitate this is to use JavaScript to cause the client to connect repeatedly to the victim server, as described in Section 7.3.1. Each connection is tested against the oracle with only small number of fractions, and the attacker can discern immediately when he receives a positive response from the oracle.

Once the attacker has obtained a positive response, he can proceed to the final phase of the special DROWN attack described above, which employs 200-bit rotation 10 times to fully decrypt the plaintext. Our current implementation requires under 30 seconds for this phase on a single PC.

The ability of the victim server to perform 17,000 handshakes in less than a minute is not an impediment for modern hardware. An RSA private key operation with a 2048-bit modulus requires on the order of 1 ms using OpenSSL on a recent-generation CPU, so the cryptographic portion of the attacker's queries induces additional server load of roughly 14 core-seconds. In tests with a nearby server running Apache 2.4, we could easily complete 10,000 HTTPS requests in under 10 seconds.

7.6. Measurements

We performed Internet-wide scans to analyze the number of systems vulnerable to DROWN. A host is directly vulnerable to general DROWN if it supports SSLv2. Similarly, a host is

directly vulnerable to special DROWN if it supports SSLv2 and has the extra clear bug. These directly vulnerable hosts can be used as oracles to attack any other host with the same key. Hosts that do not support SSLv2 are still vulnerable to general or special DROWN if their RSA key pair is exposed by any general or special DROWN oracle, respectively. The oracles may be on an entirely different host or port. Additionally, any host serving a browser-trusted certificate is vulnerable to a special DROWN man-in-the-middle if any name on the certificate appears on any other certificate containing a key that is exposed by a special DROWN oracle.

We used ZMap [114] to perform full IPv4 scans on eight different ports during late January and February 2016. We examined port 443 (HTTPS), and common email ports 25 (SMTP with STARTTLS), 110 (POP3 with STARTTLS), 143 (IMAP with STARTTLS), 465 (SMTPS), 587 (SMTP with STARTTLS), 993 (IMAPS), and 995 (POP3S). For each open port, we attempted three complete handshakes: one normal handshake with the highest available SSL/TLS version; one SSLv2 handshake requesting an export RC2 cipher suite; and one SSLv2 handshake with a non-export cipher and sixteen bytes of plaintext key material sent during key exchange, which we used to detect if a host has the extra clear bug.

We summarize our general DROWN results in Table 36. The fraction of SSL/TLS hosts that directly supported SSLv2 varied substantially across ports. 28% of SMTP servers on port 25 supported SSLv2, likely due to the opportunistic encryption model for email transit. Since SMTP fails-open to plaintext, many servers are configured with support for the largest possible set of protocol versions and cipher suites, under the assumption that even bad or obsolete encryption is better than plaintext [77]. The other email ports ranged from 8% for SMTPS to 20% for POP3S and IMAPS. We found 17% of all HTTPS servers, and 10% of those with a browser-trusted certificate, are directly vulnerable to General DROWN.

Widespread public key reuse. Reuse of RSA key material across hosts and certificates is widespread, as has been documented in [169, 216]. In many cases this is benign: many

organizations issue multiple TLS certificates for distinct domains (e.g. one for each TLD) with the same public key; reusing the same key simplifies the use of SSL acceleration hardware and load balancing. However, there is also evidence that system administrators may not entirely understand the role of the public key in certificates. For example, in the wake of the Heartbleed vulnerability, a substantial fraction of compromised certificates were reissued with the same public key [116].

There are many reasons why the same public key or certificate would be reused across different ports and services within an organization. For example a mail server that serves SMTP, POP3, and IMAP from the same daemon would likely share the same TLS configuration. Additionally, an organization might choose to purchase a single wildcard TLS certificate, and use it on both web servers and mail servers. Public keys have also been observed to be widely shared across independent organizations due to default certificates and public keys that are shipped with networked devices and software, improperly configured virtual machine images, and random number generation flaws.

The number of hosts vulnerable to DROWN rises significantly when we take RSA key reuse into account. For HTTPS, 17% of hosts are vulnerable to general DROWN because they support both TLS and SSLv2 on the HTTPS port, but the number of vulnerable hosts rises to 33% when considering RSA keys used by another service that is vulnerable to DROWN. Appendix 7.A gives more detailed statistics on the reuse of RSA key material across hosts and ports.

Special DROWN. As shown in Table 37, 9.1 M HTTPS servers (26%) are vulnerable to special DROWN, as are 2.5 M HTTPS servers with browser-trusted certificates (14%). 66% as many HTTPS hosts are vulnerable to special DROWN as to general DROWN (70% for browser-trusted servers). While there are 2.7 M public keys that are vulnerable to general DROWN, we find 1.1 M vulnerable to special DROWN (41% as many). Vulnerability among Alexa Top Million domains is lower, with only 9% of Alexa domains vulnerable (7% for browser-trusted domains).

Since special DROWN enables active man-in-the-middle attacks, any host serving a browser-trusted certificate with at least one name that appears on any certificate with a key exposed by a special DROWN oracle is vulnerable to a impersonation attacks. Extending our search to account for shared names, we find 3.8 M (22%) of hosts with browser-trusted certificates are vulnerable to man-in-the-middle, as well as 19% of the browser-trusted Alexa Top Million.

7.7. Signature forgery attacks and QUIC

An attacker can also use a Bleichenbacher-type attack to compute valid RSA signatures on arbitrary messages. Mathematically, RSA signing and decryption are identical. Such an attack could theoretically be used to forge a signed Server Key Exchange message for Diffie-Hellman cipher suites, thus allowing an attacker to perform a man-in-the-middle attack against all TLS versions up to TLSv1.3. [180] Since the server key exchange message includes the client and server randoms, the attacker must forge the signature online before the handshake times out. We are not able to use all of our optimizations for signature forgery, so such an attack does not seem feasible without additional improvements, even for special DROWN.

7.7.1. Extending the attack to QUIC

However, our attack can be extended to a feasible-time man-in-the-middle attack against QUIC [180]. QUIC [89, 272] is a recent cryptographic protocol designed and implemented by Google that is intended to reduce the setup time to establish a secure connection while providing security guarantees analogous to TLS. QUIC's security relies on a static "server config" message signed by the server's public key. Jager et al. [180] observe that an attacker who can forge a signature on a malicious QUIC server config once would be able to impersonate the server indefinitely. In this section, we show an attacker with significant resources would be able to successfully mount such an attack against a server who exposed their RSA public keys via SSLv2.

A QUIC client receives a "server config" message enumerating connection parameters, a static elliptic curve Diffie-Hellman public value, and a validity period that is signed by the

server's public key. An attacker could generate a Diffie-Hellman public value for which he knows the private key, and set the expiration date far in the future in order to mount a man-in-the-middle attack against any client.

Unauthenticated QUIC discovery. In order to mount the attack, the attacker needs to present a forged QUIC config to the client. This is straightforward, since QUIC discovery may happen over non-encrypted HTTP [162]. The server does not even need to support QUIC at all: an attacker could impersonate the attacked server over an unencrypted connection and falsely indicate that the server supports QUIC. The next time the client connects to the server, it will attempt to connect using QUIC, allowing the attacker to present the forged "server config" message and execute the attack. [180]

Signature forgery details. The attack proceeds much as in Section 7.3.3, except that we are not able to use some of the optimizations so it is more expensive.

The first step is to discover a valid, PKCS conformant SSLv2 ciphertext. In the case of TLS decryption, our input ciphertext was PKCS conformant to begin with; this is not the case for our QUIC message c_0. Thus for the first phase, we iterate through possible multiplier values s until the attacker randomly encounters a valid SSLv2 message in $c_0 \cdot s$. For 2048-bit RSA keys, the probability of this random event is $P_{rnd} \approx 2^{-25}$; see Section 7.3.3 for the computation.

Once the first SSLv2 conformant message is found, the attacker proceeds with the signature forgery as he would in Step 2 of the attack against TLS. The required number of oracle queries for this step is roughly 12,468 for 2048-bit RSA keys.

Attack cost. The overall oracle query cost is dominated by the $2^{25} = 34$ million expected queries in the first phase, above. At a rate of 388 queries/second, an attacker would finish in one day; at a rate of 12 queries/second an attacker would finish in one month.

For the SSLv2 export padding oracle, the offline computation to break a 40-bit symmetric

key for each query requires iterating over 2^{65} keys. At our optimized GPU implementation rate of 515 million keys per second, this would require 829,142 GPU days. Our experimental GPU hardware retails for $400. An investment of $10 million to purchase 25,000 GPUs would reduce the wall clock time for the attack to 33 days. Our implementation run on Amazon EC2 processed about 174 billion keys per `g2.2xlarge` instance-hour, so at a cost of $0.09/instance-hour the full attack would cost $9.5 million dollars and could be parallelized to Amazon's capacity.

For the extra clear oracle, there is only negligible computation per oracle query, so the computational cost for the first phase is 2^{25}.

Future changes to QUIC. In addition to disabling QUIC support for non-whitelisted servers, Google have informed us that they plan to change the QUIC standard, so that the "server config" message will include a client nonce to prove freshness. They also plan to limit QUIC discovery to HTTPS.

7.7.2. SSLv2 servers with CA certificates

Some web servers support SSLv2 while presenting a CA certificate, which can be used to issue further leaf certificates. In that case, an attacker could create his own certificate and use the vulnerable server to forge a CA signature over his certificate by executing an attack similar to the above. The number of queries is identical to the number of queries required for the attack against QUIC. This attack would allow the attacker to impersonate any website against any client trusting the CA certificate.

We did not observe any trusted CA certificates used on vulnerable servers. We did, however, observe a number of routers that supported SSLv2 while presenting CA certificates that are untrusted by modern browsers.

7.8. Related work

Bleichenbacher's attack. Bleichenbacher's adaptive chosen ciphertext attack against SSL was first published in 1998 [73]. Since then, several works have adapted his attack to different scenarios [48, 177, 197].

Despite the fact that the TLS standard [106] explicitly introduces countermeasures against Bleichenbacher's attack, several modern implementations have been discovered to be vulnerable to it in recent years. Meyer *et al.* [221] inspected various software and hardware implementations and discovered timing side-channels that enabled the attack. Zhang *et al.* applied Bleichenbacher's attack to develop a cache flush-and-reload timing attack against OpenSSL in cross-tenant environments [317]. These side-channel attacks, however, are applicable only in scenarios where the attacker is physically close to or co-located with the victim and are based on implementation failures.

Jager et al. described a similar Bleichenbacher oracle, as we use in our paper, to attack XML Encryption in Web Services [177]. To this end, they exploited the fact that RSA PKCS#1 v1.5 was used in combination with symmetric algorithms in CBC mode of operation.

Cross-protocol attacks. Jager et al. [180] observed that a cross-protocol Bleichenbacher RSA padding oracle attack is possible against the proposed TLSv1.3 standard, in spite of the fact that TLSv1.3 does not include RSA key exchange, if server implementations use the same certificate for previous versions of TLS and TLSv1.3. Wagner and Schneier [302] developed a cross-cipher suite attack for SSLv3, in which an attacker could reuse a signed server key exchange message in a later exchange with a different cipher suite. Mavrogiannopoulos et al [215] developed a cross-cipher suite attack allowing an attacker to use elliptic curve Diffie-Hellman as plain Diffie-Hellman.

Attacks on export-grade cryptography. Recently, the FREAK [61] and Logjam [29] attacks allowed an active attacker to downgrade a connection to export-grade RSA and Diffie-Hellman, respectively. Export-grade cryptography plays an important role in DROWN as well, as it exploits export-grade symmetric ciphers.

Further attacks on SSL/TLS. Other attacks on SSL and TLS include: POODLE [225], which exploits SSLv3's lack of a requirement for the contents of padding bytes, and its

MAC-then-encrypt construction; CRIME [271], which exploits support for compression and observes ciphertexts' lengths in order to decrypt traffic; The RC4 Biases attack [33], which utilizes biases in the the RC4 keystream; Lucky13 [32], which exploits small timing differences and MAC-then-encrypt; and BEAST [112], which exploits predictable IVs in TLS. Bhargavan and Leurent presented SLOTH attacks and broke TLS and other protocols using MD5 for computing transcript hashes [63].

7.9. Discussion

7.9.1. Lessons for protocol design

A natural question is to ask whether SSLv3 or later versions of TLS could also be vulnerable. Our attack exploits two properties of the SSLv2 protocol:

Server authenticates first. First, the fact that in SSLv2 the server responds to the `ClientMasterKey` message before the client proves it has knowledge of the RSA plaintext, provides a direct message side channel. In SSLv3 and later, the client must demonstrate knowledge of the RSA plaintext first via a valid `ClientFinished` message before the server sends a message derived from the RSA plaintext. In order to perform a similar attack in this case, the client would need to perform an online brute-force attack.

Short secrets. Second, SSLv2 allows RSA plaintexts that are short enough to be vulnerable to a feasible-time brute force search. For export ciphers, the unpadded RSA plaintext is five bytes long. In SSLv3 and later versions of TLS, the RSA plaintexts and premaster secret length is 48 bytes, even for export ciphers with 40-bit strength. For later protocol versions, an attacker can perform a brute-force search over the derived 40-bit key if a client negotiates an export cipher suite, but the 48-byte premaster secret length appears to prevent an attacker from escalating the weakness of the export cipher strength into a similar protocol vulnerability.

7.9.2. Implications for modern protocols

Modern TLS versions are not vulnerable to the precise attack given in this paper, but they have similar properties that might allow a related attack.

Although we do not present concrete attacks on modern protocols, we argue that modern practices of cryptographic protocol design do not include a systematic analysis to prevent direct message side channel Bleichenbacher attacks. A hypothesized protocol with modern parameters would be vulnerable to such an attack if it has the following properties:

1. RSA key exchange. TLSv1.2 [106] allows this.
2. It allows re-use of server-side nonce by the client. QUIC [272] allows this.
3. The server sends the first message encrypted using a key derived from the asymmetric key exchange. QUIC, TLSv1.3 [107], and TLS False Start [201] exhibit this property.

When all three properties are combined, a natural adaptation of our attack presents itself. The attacker obtains a Bleichenbacher oracle by connecting to the server twice with the same RSA ciphertext and the same server-side nonce, and comparing the messages sent by the server. If the RSA ciphertext is PKCS conformant, the two messages will be identical. Otherwise, they will differ. Note that we also assumed that all symmetric cipher parameters, including IVs for block ciphers, are deterministically generated from the premaster secret and nonces; this is the case for TLSv1.0. If that is not the case, in most realistic configurations, the attacker can choose a stream cipher.

An attacker can use False Start to cause a victim client to perform TLS handshakes using RSA for key exchange, even if the server supports other key exchange methods which provide Perfect Forward Secrecy. The attacker masquerades as the server and indicates support for RSA key exchange only. The client will then handshake using RSA, and send application layer data, before the server authenticates by sending the `Finished` message. The False Start standard indeed discourages the use of RSA as the key exchange method, but does not explicitly forbid it, leaving the security of the protocol dependent on correct choices in the client configuration. Our attacks show that relying on such assumptions is extremely brittle protocol design.

7.9.3. Lessons for key reuse

Our attacks also illustrate another important cryptographic principle: that keys should be single use. For public keys we think of this principle as applying primarily to keys that are used to both sign and decrypt, but our attacks illustrate that using keys *for different protocol versions* can also be a serious security risk. Unfortunately, the TLS certificate authority funding model produces a financial incentive for users to purchase as few certificates as necessary to protect their infrastructure. However, even without this financial incentive in place, the sheer number of SSL/TLS protocol versions in use would make key management difficult.

7.9.4. Harms from obsolete cryptography

Recent years have seen a significant number of serious attacks exploiting outdated and obsolete cryptography. Many of these protocols and cryptographic primitives are surprisingly common in deployed systems even decades after they were demonstrated to be weak.

The attack described in this paper exploits a modification of an 18-year-old attack against a combination of protocols and ciphers that have long been superseded by better options: the SSLv2 protocol, export cipher suites, and PKCS #1 v1.5 RSA padding. In fact, support for RSA as a key exchange method, including the use of PKCS #1 v1.5, is mandatory even for TLSv1.2. The attack is made more severe by implementation flaws in rarely-used code.

Our work serves as yet another reminder of the importance of removing deprecated technologies before they become exploitable vulnerabilities. In response to many of the vulnerabilities listed above, browser vendors have been aggressively warning end users when TLS connections are negotiated with unsafe cryptographic parameters, including SHA-1 certificates, small RSA and Diffie-Hellman parameters, and SSLv3 connections. This process is currently happening in a piecemeal fashion, primitive by primitive. Vendors and developers rightly prioritize usability and backward compatibility, and are willing to sacrifice these only for practical attacks. This standard works less well for cryptographic vulnerabilities, where the first sign of a weakness, while far from being practically exploitable, can signal

trouble in the future. Communication issues between academic researchers and vendors and developers have been voiced by many in the community, including Green [211] and Jager et al. [178].

The long-term solution is to proactively remove these obsolete technologies. There has been a movement towards this already: TLSv1.3 has removed RSA key exchange entirely and has restricted Diffie-Hellman key exchange to a few groups large enough to withstand cryptanalytic attacks long in the future. The CA/Browser forum will remove support for SHA-1 certificates this year. And resources such as the SSL Labs SSL Reports have gathered information about best practices and vulnerabilities in one place, in order to encourage administrators to make the best choices.

7.9.5. Harms from deliberately weakening cryptography

Export-grade cipher suites for TLS deliberately weakened three primitives to the point that they are broken even to enthusiastic amateurs today: 512-bit RSA key exchange, 512-bit Diffie-Hellman key exchange, and 40-bit symmetric encryption. All three deliberately-weakened primitives have been cornerstones of high-profile attacks: FREAK attack against export RSA, Logjam against Diffie-Hellman, and our DROWN attack against export-grade symmetric cryptography.

Our results illustrate, like FREAK and Logjam, the continued harm that a legacy of deliberately weakened export-grade cryptography inflicts on the security of modern systems, even decades after the regulations influencing the original design were lifted. The attacks described in this paper are fully feasible against export cipher suites today; against even DES they would be at the limits of the computational power available to an attacker. The technical debt induced by cryptographic "front doors" has left implementations vulnerable for decades. Together with the slow rate at which obsolete protocols and primitives entirely disappear, we can expect some fraction of hosts to continue to be vulnerable for years to come.

7.A. Public key reuse

Reuse of RSA keys among different services was identified as a huge amplification to the number of services vulnerable to DROWN. Table 38 describes the number of reused RSA keys among different protocols. The two clusters 110-143 and 993-995 stick out as they share the majority of public keys. This is expected, as most of these ports are served by the same IMAP/POP3 daemon. The rest of the ports also share a substantial fraction of public keys, usually between 21% and 87%. The numbers for HTTPS (port 443) differ as there are four times as many public keys in HTTPS as in the second largest protocol.

Port	25 (SMTP)	110 (POP3)	143 (IMAP)	443 (HTTPS)	465 (SMTPS)	587 (SMTP)	993 (IMAPS)	995 (POP3S)
25	1,115 (100%)	331 (32%)	318 (32%)	196 (4%)	403 (47%)	307 (48%)	369 (33%)	321 (32%)
110	331 (30%)	1,044 (100%)	795 (79%)	152 (3%)	337 (39%)	222 (35%)	819 (72%)	877 (87%)
143	318 (29%)	795 (76%)	1,003 (100%)	149 (3%)	321 (38%)	220 (35%)	878 (78%)	755 (75%)
443	196 (18%)	152 (15%)	149 (15%)	4,579 (100%)	129 (15%)	94 (15%)	175 (16%)	151 (15%)
465	403 (36%)	337 (32%)	321 (32%)	129 (3%)	857 (100%)	463 (73%)	396 (35%)	364 (36%)
587	307 (28%)	222 (21%)	220 (22%)	94 (2%)	463 (54%)	637 (100%)	259 (23%)	229 (23%)
993	369 (33%)	819 (78%)	878 (88%)	175 (4%)	396 (46%)	259 (41%)	1,131 (100%)	859 (85%)
995	321 (29%)	877 (84%)	755 (75%)	151 (3%)	364 (42%)	229 (36%)	859 (76%)	1,010 (100%)

Table 38: **Impact of key reuse across ports**—Number of shared public keys among two ports, in thousands. Each column states what number and percentage of keys from the port in the header row are used on other ports. For example, 18% of keys used on port 25 are also used on port 443, but only 4% of keys used on port 443 are also used on port 25.

7.B. Adaptations to Bleichenbacher's attack

7.B.1. Calculating the success probability of a fraction

For a given fraction u/t, we can compute the probability of success with a randomly chosen TLS conformant ciphertext. Let $m_1 = m_0 \cdot u/t = m_1[1]||...||m_1[\ell]$ - i.e. $m_1[i]$ is the ith byte of m_1. Let k be the fixed byte length of the oracle response. For $s = u/t \bmod N$ where u and t are coprime, m_1 will be SSLv2 conformant if the following conditions all hold:

1. m_0 is divisible by t. For randomly generated m_0, this condition holds with probability $1/t$.

2. $m_1[1] = 0$ and $m_1[2] = 2$, or the integer $m \cdot u/t \in [2B, 3B - 1)$. For a randomly generated m_0 divisible by t and for a given fraction u/t, this condition holds with

probability

$$P = \begin{cases} 3 - 2 \cdot t/u & \text{for } 2/3 < u/t < 1 \\ 3 \cdot t/u - 2 & \text{for } 1 < u/t < 3/2 \\ 0 & \text{otherwise} \end{cases}$$

3. $\forall i \in [3, \ell - (k+1)], m_1[i] \neq 0$, or all bytes between the first two bytes and the $(k+1)$ least significant bytes are non-zero. This condition holds with probability $(1 - 1/256)^{\ell-(k+3)}$.

4. $m_1[\ell - k] = 0$, or the $(k+1)$st least significant byte is 0. This condition holds with probability $1/256$.

As an example, let us assume a 2048-bit RSA ciphertext with $k = 5$, and consider the fraction $u = 7, t = 8$. We have

$$\begin{aligned} P(t|m_0) &= 1/t = 1/8 \\ P(m_1[1,2] = 00||02 \,\big|\, t|m_0) &= 0.71 \\ P(\forall i \in [3, \ell - 6]\, m_1[i] \neq 0) &= (1 - 1/256)^{248} = 0.37 \\ P(m_1[\ell - 5] = 0) &= 1/256 \end{aligned}$$

The overall probability of success is $P = 1/8 \cdot 0.71 \cdot 0.37 \cdot 1/256 = 1/7,774$; thus we expect to find an SSLv2 conformant ciphertext after testing 7,774 randomly chosen TLS conformant ciphertexts. We can decrease the number of TLS conformant ciphertexts needed by multiplying each candidate ciphertext by several fractions.

7.B.2. Optimizing the chosen set of fractions

In order to deduce the validity of a single ciphertext, the attacker would have to perform a non-trivial brute-force search over all 5 byte `master_key` values. This translates into 2^{40} encryption operations.

The search space can be reduced by an additional optimization, which relies on the fractional

multipliers used in the first step. Suppose the attacker uses a fraction $u/t = 8/7$ to compute a new SSLv2 conformant candidate, and suppose that m_0 is indeed divisible by $t = 7$. This implies that the new candidate message $m_1 = m_0/t \cdot u$ is divisible by $u = 8$, and the last three bits of m_1 (and thus mk_{secret}) are zero. This allows the attacker to reduce the searched `master_key` space by selecting specific fractions.

More generally, for an integer u, the largest power of 2 by which u is divisible, is denoted by $v_2(u)$, and multiplying by a fraction u/t saves us a factor of $v_2(u)$ in the required encryption attempts. With this observation, the trade-off between the 3 metrics: the required number of intercepted ciphertexts, the required number of queries, and the required number of encryption attempts, becomes non-trivial to analyze.

Therefore, we have resorted to using simulations when evaluating the performance metrics for sets of fractions. The probability that multiplying a ciphertext by any fraction out of a given set of fractions results in an SSLv2 conformant message is difficult to compute, since the events are in fact inter-dependent: If $m \cdot 16/15$ is conforming, then m is divisible by 5, greatly increasing the probability that $m \cdot 4/5$ is also conforming. However, it is easy to perform a Monte Carlo simulation, where we randomly generate ciphertexts, and measure the probability that any fraction out of a given set produces a conforming message. The expected required number of intercepted ciphertexts is the inverse of that probability.

Formally, if we denote the set of fractions as F, and the event that a message m is conforming as $C(m)$, we perform a Monte Carlo estimation of the probability $P_F = P(\exists f \in F : C(m \cdot f))$, and the expected number of required intercepted ciphertexts equals $1/P_F$.

The required number of oracle queries is simply $1/P_F \cdot |F|$: For each ciphertext, we need to query the oracle with each fraction. Accordingly, the required number connections to the server is $2 \cdot 1/P_F \cdot |F|$, since as explained earlier each logical query consists of two connections to the server.

And as for the required number of encryption attempts, if we denote this number when

querying with a given fraction $f = u/t$ as E_f, then $E_f = E_{u/t} = 2^{40-v_2(u)}$. If we further define the required encryption attempts when testing a single ciphertext with each fraction from a given set of fraction F as $E_F = \sum_{f \in F} E_f$ then the required number of encryption attempts throughout the attack for a given set of fractions is $(1/P_F) \cdot E_F$.

Using this approach, we can now give precise figures for the expected number of required intercepted ciphertexts, connections to the targeted server, and encryption attempts. The results presented in Table 34 were obtained by using the monte-carlo estimation technique described above, with one billion random ciphertexts per tested fraction set F.

7.B.3. Efficiently computing rotations and multipliers

For a randomly chosen s, the probability that the two most significant bytes are `0x00 02` is 2^{-16}; for a 2028-bit modulus N the probability that the next $\ell - k - 3$ bytes of m_2 are all nonzero is about 0.37 as in the previous section, and the probability that the $k + 1$ least significant delimiter byte is `0x00` is 1/256. Thus a randomly chosen s will work with probability $2^{-25.4}$ and we expect to need to try $2^{25.4}$ values of s before succeeding.

However, since we have already learned $k+3$ most significant bytes of $m_1 \cdot R^{-1} \bmod N$, for $k \geq 4$ and $s < 2^{30}$ we do not need to query the oracle to learn if the two most significant bytes are SSLv2 conformant; we can compute this ourselves from our knowledge of $\tilde{m_1} \cdot R^{-1}$. We could simply iterate through values of s, test that the top two bytes of $\tilde{m_1} \cdot R^{-1} \bmod N$ are SSLv2 conformant, and only query the oracle $\mathcal{O}$ for values of s that satisfy this test; this means that for our 2048-bit modulus we expect to test 2^{16} values offline per oracle query. The probability that our query is conformant is then $P = (1/256) * (255/256)^{249} \approx 1/678$ so we expect to perform 678 oracle queries before finding a fully SSLv2 conformant ciphertext $c_2 = (s \cdot R^{-1})^e c_1 \bmod N$.

We can speed up the brute force testing of 2^{16} values of s using algebraic lattices. We are searching for values of s satisfying $\tilde{m_1} R^{-1} s < 3B \bmod N$, or given an offset s_0 we would like to find solutions x and z to the equation $\tilde{m_1} R^{-1}(s_0 + x) = 2B + z \bmod N$ where $|x| < 2^{16}$

and $|z| < B$. Let $X = 2^{15}$. We can construct the lattice basis

$$L = \begin{bmatrix} -B & X\tilde{m}_1 R^{-1} & \tilde{m}_1 R^{-1} s_0 + B \\ 0 & XN & 0 \\ 0 & 0 & N \end{bmatrix}$$

We then run the LLL algorithm [203] on L to obtain a reduced lattice basis V containing vectors v_1, v_2, v_3. We then construct the linear equations $f_1(x, z) = v_{1,1}/B \cdot z + v_{1,2}/X \cdot x + v_{1,3} = 0$ and $f_2(x, z) = v_{2,1}/B \cdot z + v_{2,2}/X \cdot x + v_{2,3} = 0$ and solve the system of equations to find a candidate integer solution $x = \tilde{s}$. We then test $s = \tilde{s} + s_0$ as our candidate solution in this range.

$\det L = XZN^2$ and $\dim L = 3$, thus we expect the vectors v_i in V to have length approximately $|v_i| \approx (XZN^2)^{1/3}$. We will succeed if $|v_i| < N$, or in other words $XZ < N$. $N \approx 2^{8\ell}$, so we expect to find short enough vectors. This approach works well in practice and is significantly faster than iterating through 2^{16} possible values of $\tilde{s}$ for each query.

In summary, given an SSLv2 conformant ciphertext $c_1 = m_1^e \bmod N$, we can efficiently generate an SSLv2 conformant ciphertext $c_2 = m_2^e \bmod N$ where $m_2 = s \cdot m_1 \cdot R^{-1} \bmod N$ and we know several most significant bytes of m_2, using only a few hundred oracle queries in expectation. We can iterate this process as many times as we like to continue generating SSLv2 conformant ciphertexts c_i for which we know increasing numbers of most significant bytes, and which have a known multiplicative relationship to our original message c_0.

7.B.4. Rotations in the general DROWN attack

After the first phase, we have learned an SSLv2 conformant ciphertext c_1, and we wish to shift known plaintext bytes from least to most significant bits. Since we learn the least significant 6 bytes of plaintext of m_1 from a successful oracle $\mathcal{O}_{\mathsf{SSLv2\text{-}export}}$ query, we could use a shift of 2^{-48} to transfer 48 bits of known plaintext to the most significant bits of a new ciphertext. However, we perform a slight optimization here, to reduce the number of encryption attempts. We instead use a shift of 2^{-40}, so that the least significant byte

of $m_1 \cdot 2^{-40}$ and $\tilde{m_1} \cdot 2^{-40}$ will be known. This means that we can compute the least significant byte of $m_1 \cdot 2^{-40} \cdot s \bmod N$, so oracle queries now only require 2^{32} encryption attempts each. This brings the total expected number of encryption attempts for this phase to $2^{32} * 678 \approx 2^{41}$.

We perform two such plaintext shifts in order to obtain an SSLv2 conformant message, m_3 that resides in a narrow interval of length at most $2^{8\ell-66}$. Then we can then obtain a multiplier s_3 such that $m_3 \cdot s_3$ is also SSLv2 conformant. Since m_3 lies in an interval of length is at most $2^{8\ell-66}$, with high probability for any $s_3 < 2^{30}$, $m_3 \cdot s_3$ lies in an interval whose length is at most $2^{8\ell-36} < B$, so we know the two most significant bytes of $m_3 \cdot s_3$. Furthermore, we know the exact value of the 6 least significant bytes even after multiplication. So we test possible values of s_3, and for values such that $m_3 \cdot s_3$ starts with the required 00 02 bytes, and the 6th least significant byte is zero, we query the oracle as to the validity of $c_3 \cdot s_3^e \bmod N$. The only condition for PKCS conformance which we haven't verified before querying the oracle is

$$\forall i \in [3, \ell - 6], (m_3 \cdot s_3)[i] \neq 0$$

which holds with probability 0.37. So after roughly $1/0.37 = 2.72$ queries, we expect to get a positive answer from the oracle.

Since we know the value of the 6 least significant bytes after multiplication, there's no component of breaking a symmetric cipher here - if the message is SSLv2 conformant after multiplication, we know the symmetric key, and can test whether it fits the received `ServerVerify` message.

7.B.5. General DROWN Bleichenbacher iterations

After we have bootstrapped the attack using rotations,, the original algorithm proposed by Bleichenbacher can be applied with minimal modifications.

The original step obtains a message that starts with the required 00 02 bytes once in roughly

every two queries on average, and requires the number of queries to be roughly double the number of bits in the RSA modulus. Since we know the value of the 6 least significant bytes after multiplying by any integer, we can only query the oracle for multipliers that cause the 6th least significant byte to be zero, and we don't need to break a symmetric key since we know the value of the 5 least significant bytes. However, we cannot ensure that the padding is non-zero when querying—we simply hope that is the case, which as usual happens with probability 0.37.

Therefore, for a 2048-bit modulus, the overall expected number of queries for this phase is roughly $2048 * 2/0.37 = 11,000$. This is indeed the average number of queries we require in practice when running our implementation of the attack.

7.B.6. General DROWN attack performance

For a given set of fractions, F, the required number of recorded client connections A is a random variable distributed geometrically with a success probability $P = P_F$. For typical fraction sets, $1/13,000 < P_F < 1/600$. The required number of Bleichenbacher queries against the target server during the first step of the attack is a random variable, B, such that $B = |F| \cdot A$. As each query consists of two separate connections to the target server, the required number of connections is always twice the number of queries. And last, the required keys to be tested overall is another random variable $C = k_F \cdot B; k_F \approx 2^{40}$.

Summing the figures from the different phases for a 2048-bit RSA modulus, the attack requires in expectation $13,838 + 1,393 + 1,393 + 6 + 22,140 = 38,770$ connections to the target server, when optimizing for the number of queries in phase 1. Each oracle query requires two connections to the server.

Re-calculating the numbers for a 1024 bit modulus, the primary element that needs to change is $P_1 = P(\forall i \in [3, \ell - 6] : m_i \neq 0) = (1 - 1/256)^{120} = 0.62$, which appears in phases 1, 2, 3 and 5. For phase 5, the number of queries is now in expectation $1024 * 2/0.62 = 3,303$. The total expected number of server connections is therefore $8,258 + 826 + 826 + 6 + 6,606 = 16,522$, again when optimizing for the number of queries in phase 1.

Similarly, re-calculating the numbers for a 4096 bit modulus, $P_1 = (1-1/256)^{504} = 0.14$, and the number of queries in phase 5 is now roughly $4096 * 2/0.14 = 58,514$. The algorithm for phase 5 can be further optimized if that is the case of interest; we omit these optimizations for space reasons. Again, summing up yields $36,571+3,657+3,657+29+117,028 = 160,942$ required connections to the server.

7.B.7. Special DROWN attack performance

In the first step, we can use the same fraction analysis as before. The probability that the three padding bytes are correct remains unchanged. The probability that all the intermediate padding bytes are non-zero is now slightly higher, $P_1 = (1 - 1/256)^{229} = 0.41$, yielding an overall maximal success probability $P = 0.1 \cdot 0.41 \cdot \frac{1}{256} = 1/6,244$ per oracle query. Since we now only need to connect to the server once per oracle query, the expected number of connections in this step is the same, $6,243$. Phase 1 now yields a message with 3 known padding bytes and 24 known plaintext bytes.

For the remaining rotation steps, each rotation requires an expected 630 oracle queries. The attacker at this point could directly complete the original Bleichenbacher attack by performing 11,000 sequential queries in the final phase. However, with this more powerful oracle it is more efficient for the attacker to apply a rotation 10 more times to recover the remaining bits of the plaintext. The number of queries required in this phase is now $10 \cdot 256/0.41 \approx 6,300$, and the queries for each of the 10 steps can be executed in parallel.

Using multiple queries per fraction. For the $\mathcal{O}_{\mathsf{SSLv2\text{-}extra\text{-}clear}}$ oracle, the attacker can increase his chances of success by querying the server multiple times per ciphertext and fraction, using different cipher suites with different key lengths. He can query DES and hope the 9th least significant byte is zero, then negotiate 128-bit RC4 and hope the 17th least significant byte is zero, then negotiate 3DES and hope the 25th least significant is zero. All three queries also require the intermediate padding bytes to be non-zero. This technique triples the success probability for a given pair of (ciphertext, fraction), at a cost of triple the queries. Its primary benefit is that fractions with smaller denominators (and

Platform	Hardware	Cost	Full attack	Cost to perform attack in 1 day
Naïve CPU	4 Intel Xeon E7-4820	$21,400	114 days	$2,440,000
Naïve GPU	ZOTAC GeForce GTX TITAN	$2,400	189 days	$450,000
Naïve FPGA	64 Spartan-6 LX150	$60,000	51.5 days	$3,090,000
Optimized Hashcat	NVIDIA GTX / AMD R9	$18,040	0.75 days	$13,500
Optimized EC2	NVIDIA	$440	0.33 days	$147

Table 39: **Time and cost efficiency of our attack on different hardware platforms**—The brute force attacks against symmetric export keys are the most expensive part of our attack. We compared the performance of a naïve implementation of our attack on different platforms, and decided that a GPU implementation held the most promise. We then heavily optimized our GPU implementation, obtaining several orders of magnitude in speedup.

thus higher probabilities of success) are now even more likely to succeed.

For a random ciphertext, when choosing 70 fractions, the probability of the first zero delimiter byte being in one of these three positions is 0.01. Hence, the attacker can use only 100 recorded ciphertexts, and expect to use $100 * 70 * 3 = 21,000$ oracle queries. For the extra clear oracle, each query requires one SSLv2 connection to the server. After obtaining the first positive response from the oracle, the attacker proceeds to phase 2 using 3DES.

7.C. Highly optimized GPU implementation

The most computationally expensive part of our general DROWN attack is breaking the 40-bit symmetric key. We wanted to find the platform that would have the best tradeoff of cost and speed for the attack, so we performed some preliminary experiments comparing performance of symmetric key breaking on CPUs, GPUs, and FPGAs. These experiments used a naïve version of the attack using the OpenSSL implementation of MD5 and RC2.

The CPU machine contained four Intel Xeon E7-4820 CPUs with a total of 32 cores (64 concurrent threads). The GPU system was equipped with a ZOTAC GeForce GTX TITAN and an Intel Xeon E5-1620 host CPU. The FPGA setup consisted of 64 Spartan-6 LX150 FPGAs.

We benchmarked the performance of the CPU and GPU implementations over a large corpus of randomly generated keys, and then extrapolated to the full attack. For the FPGAs, we

tested the functionality in simulation and estimated the actual runtime by theoretically filling the FPGA up to 90% with the design, including communication. Table 39 compares the three platforms.

While the FPGA implementation was the fastest in our test setup, the speed-to-cost ratio of GPUs was the most promising. Therefore, we decided to focus on optimizing the attack on the GPU platform. We developed several optimizations:

Generating key candidates on GPUs. Our naïve implementation generated key candidates on the CPUs. For each hash computation, a key candidate was transmitted to the GPU, and the GPU responded with the key validity. The bottleneck in this approach was the PCI-E Bus. Even newer boards with PCI-E 3.0 or even PCI-E 4.0 are too slow to handle the large amount of data required to keep the GPUs busy. We solved this problem by generating the key candidates directly on the GPUs.

Generating memory blocks of keys. Our hash computation kernel had to access different candidate keys from the GPU memory. Accessing global memory is typically a slow operation and we needed to keep memory access as minimal as possible. Ideally we would be able to access the candidate keys on a register level or from a constant memory block, which is almost as fast as a register. However, there are not enough registers or constant memory available to store all the key values.

We decided to divide each key value into two parts k_H and k_L, where $|k_H| = 1$ byte and $|k_L| = 4$ bytes. We stored all possible 2^8 k_H values in the constant read-only memory, and all possible 2^{32} k_L values in the global memory. Next we used an in-kernel loop. We loaded the latter 4 bytes from the slow global memory and stored it in registers. Inside the inner loop we iterated through our first byte k_H by accessing the fast constant memory. The resulting key candidate was computed as $k = k_H||k_L$.

Using 32-bit data types. Although modern GPUs support several data types ranging in size from 8 to 64 bits, many instructions are designed for 32-bit data types. This fits the

design of MD5 perfectly, because it uses 32-bit data types. RC2, however, uses both 8-bit and 16-bit data types, which are not suitable for 32-bit instruction sets. This forced us to rewrite the original RC2 algorithm to use 32-bit instructions.

Avoiding loop branches. Our kernel has to concatenate several inputs to generate the `server_write_key` needed for the encryption as described in Section 7.2.2. Using loops to move this data generates branches because there is always an if() inside a for() loop. To avoid these branches, which always slow down a GPU implementation, we manually shifted the input bytes into the 32-bit registers for MD5. This was possible since the hash computation inputs, $(mk_{clear}||mk_{secret}||\text{“0”}||r_c||r_s)$, have constant length.

Optimizing MD5 computation. Our MD5 inputs have known input length and block structure, allowing us to use the so-called zero-based optimizations. Given the known input length (49 bytes) and the fact that MD5 uses zero padding, in our case the MD5 input block included four 0x00 bytes. These `0x00` bytes are read four times per MD5 computation which allowed us to drop in total 16 ADD operations per MD5 computation. In addition, we applied the Initial-step optimizations used in the Hashcat implementation [287].

Skipping the second encryption block. The input of the brute-force computation is a 16-byte client challenge r_c and the resulting ciphertext from the `ServerVerify` message which is computed with an RC2 cipher. As RC2 is an 8-byte block cipher the RC2 input is split into two blocks and two RC2 encryptions are performed. In our verification algorithm, we skipped the second decryption step as soon as we saw the key candidate does not decrypt the first plaintext block correctly. This resulted in a speedup of about a factor of 1.5.

RC2 permutation table in constant memory. The RC2 algorithm uses a 256-byte permutation table which is constant for all RC2 computations. Hence, this table is a good candidate to be put into the constant memory, which is nearly as fast as registers and makes it easy to address the table elements. When finally using the values, we copied them into the even faster shared memory. Although this copy operation has to be repeated, it still

led to a speed up of approximately a factor of 2.

RC2 key setup without keysize checks. The key used for RC2 encryption is generated using MD5, thus the key size is always 128 bits. Therefore, we do not have to check for the input key size, and can simply skip the size verification branch completely.

7.D. Amazon EC2 evaluation

Amazon EC2 billing is based on the *instance-hour*. An *instance* represents a single virtualized machine and its associated cores, memory, and storage. For our experiments we used `g2` instances, which are equipped with high-performance NVIDIA GPUs, each with 1,536 CUDA cores. The two available models for this instance type are the `g2.2xlarge` and the `g2.8xlarge`, containing one and four GPUs, respectively.

It is possible to request instances at a fixed on-demand rate, or bid on instances at the discounted spot instance rate. Spot instances may be terminated depending on demand, but the savings in cost are significant compared to the on-demand rate. When we ran our experiments in January 2016, the on-demand rate for the `g2.2xlarge` model was $0.65/hr and the rate for the `g2.8xlarge` model was $2.65/hr, while the average spot rates we paid were $0.09/hr and $0.83/hr respectively.

We used a cluster composed of 200 spot instances: 150 `g2.2xlarge` which contain one GPU and 50 `g2.8xlarge`, each containing four GPUs, spread across multiple availability zones within the US-East region. This distribution was determined by price: we were not able to launch more than 50 `g2.8xlarge` instances without a sharp spike in spot prices. We used the optimized Hashcat implementation on the same workload of key requests as the experiments run on the Hashcat servers. We used Slurm [312] to distribute jobs across compute nodes.

The GPU breaking experiment completed successfully, with two minor caveats. First, the 150 `g2.2xlarge` nodes completed their workloads at the 6h26m mark, while the other 50 `g2.8xlarge` nodes did not finish until the 7h41m mark. More careful job distribution

would ensure that all nodes completed at approximately the same time, reducing the overall runtime. Second, in this particular run, 7.2% of the jobs that we expected to complete were terminated early due to overheating GPUs. The attack was successful despite the failed jobs, so we did not rerun them. In a more carefully engineered implementation, the unfinished jobs could have been reallocated to the unused GPU capacity without increasing the overall runtime.

The total cost of the experiment was $440, and terminated in under 8 hours including startup and shutdown.

7.E. A brief history of obsolete cryptography

A flaw was first observed in the MD5 hash function in 1996; the first collision was discovered in 2004 [304], but MD5 was still in use by certificate authorities in 2009 when Stevens et al. [288] used a chosen-prefix MD5 attack to construct a malicious TLS certificate with a valid CA signature. The RC4 stream cipher was observed to be biased as early as 1995 and shown to be catastrophically broken in the context of WEP in 2001 [127]; it was used by about 50% of TLS connections in 2013 when AlFardan et al. [33] demonstrated near-practical attacks against RC4 in TLS. TLSv1.0 was standardized in 1998 to replace SSLv3; before the POODLE attack [225] was shown to render all SSLv3 block cipher suites insecure in 2014, support for SSLv3 was near 100% for popular HTTPS sites, and most clients were vulnerable to a downgrade attack from TLS to SSLv3 [269]. Export-grade cipher suites for TLS have been obsolete since 2000, when the United States relaxed restrictions on commercial and open source software; before the FREAK attack [61] demonstrated widespread implementation flaws allowing a catastrophic downgrade attack exploiting export RSA, 37% of HTTPS sites with browser-trusted certificates supported export-grade RSA. Three months later the Logjam attack [29] demonstrated a TLS protocol flaw downgrade attack exploiting export Diffie-Hellman; 8.4% of the Alexa top million sites were vulnerable at the time.

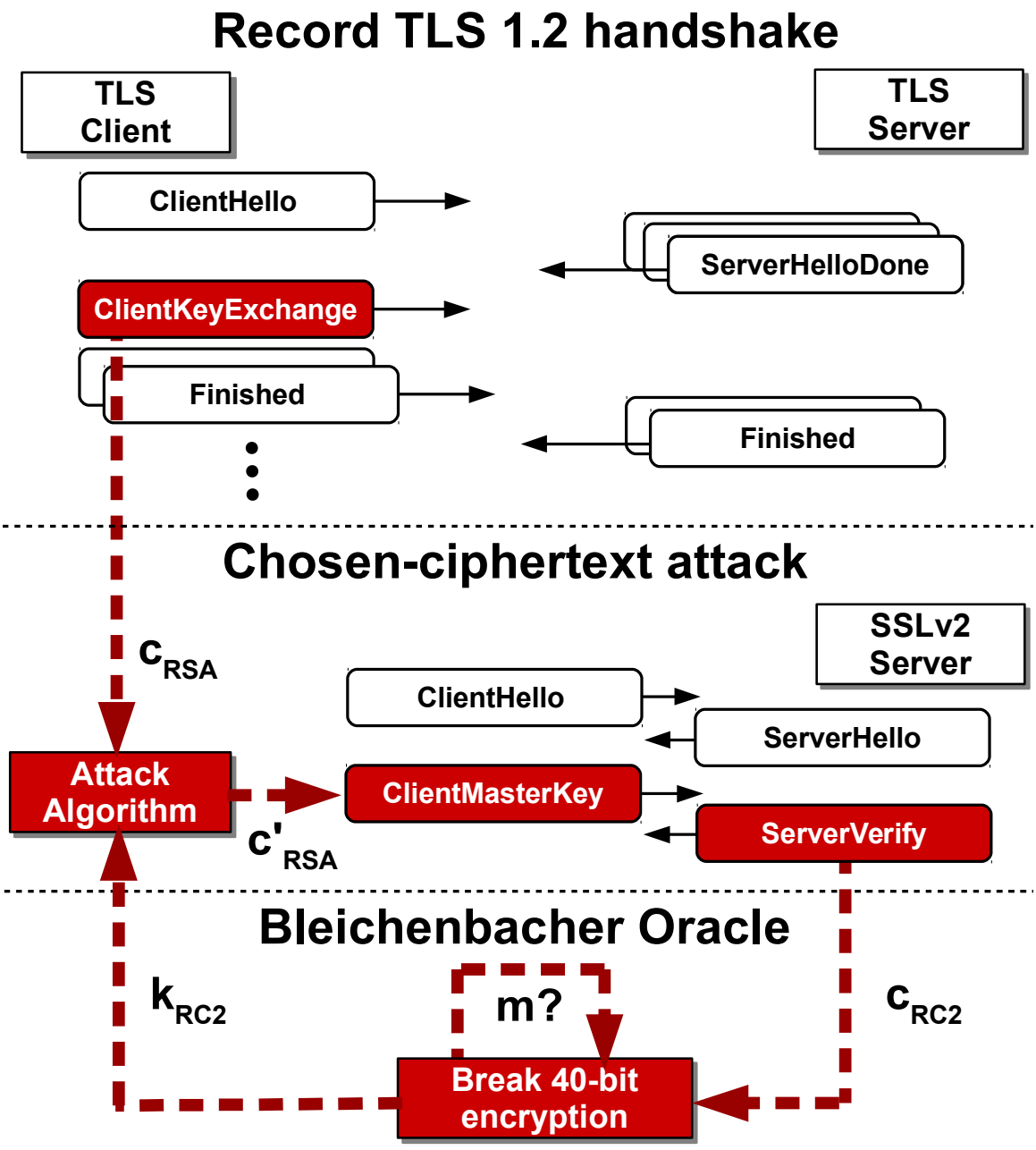


Figure 16: **Our SSLv2-based Bleichenbacher attack on TLS**—An attacker passively collects RSA ciphertexts from a TLSv1.2 handshake, and then performs oracle queries against a server that supports SSLv2 with the same public key to decrypt the TLS ciphertext.

CHAPTER 8 : Conclusion

In this dissertation, I present six case studies in which I demonstrate vulnerabilities in cryptographic deployments. I also present a framework within which I characterize the root cause of each of these vulnerabilities as a failure in communication from one stage of the cryptographic deployment process to the next, or as stemming from incentives at odds with security.

8.1. Takeaways

I now present several key takeaways from this dissertation.

Securing systems through improved cryptographic knowledge transfer. Minimizing the attack surface of a system is a standard security engineering practice to improve the security of a system. The framework presented in this dissertation demonstrates that cryptographic vulnerabilities can be characterized as a failure of knowledge transfer from one stage of the cryptographic deployment process to the next. This suggests that methods for preserving information from one stage in the process to the next can reduce the risk of future vulnerabilities.

The first approach that one could take in this direction is to minimize the security-critical decisions that need to be made by non-experts in cryptographic deployments. Many of the vulnerabilities discussed in this thesis are the result of configuration errors or performance tradeoffs made on the part of application developers or system administrators. Cryptographic library developers can help with this by simplifying APIs and providing secure-by-default configurations.

A second approach for reducing information loss is to have a single well-qualified entity simultaneously execute multiple stages of the cryptographic deployment process. For instance, Bernstein [55] presents the X25519 algorithm along with detailed guidelines for its use, an optimized and deployment-ready reference implementation, and practical performance statistics. However, this approach likely does not scale.

Other promising steps in this direction are formal verification of cryptographic standards [66, 67, 69, 109] and implementations [64], though gaps in proofs can still result in vulnerabilities [110].

Solving versus eradicating a problem. Known weaknesses in cryptographic algorithms repeatedly arise as security vulnerabilities in modern systems, despite solutions for these problems being known about (in some cases) for decades. While completely eradicating classes problems in *newly-designed* systems is possible (as demonstrated by the complete removal of RSA key exchange and all of its associated weaknesses in TLSv1.3), systems that need to maintain support for legacy protocols cannot use this approach. This observation leads to the conclusion that vulnerability mitigation should not stop at merely identifying classes of problems and developing countermeasures, but requires continual reevaluation of systems with respect to known weaknesses. The tools and methods presented in this dissertation provide ways to detect and measure classes of vulnerabilities at scale.

Internet scanning for vulnerability discovery. Fast internet-wide scanning of end hosts has become an important tool for security researchers over the past several years. A global view of servers has enabled measurement studies of protocol adoption and implementation choices, and allowed researchers to measure the impact of vulnerabilities like the Debian OpenSSL disaster [310], Heartbleed [116], and the Dual EC backdoor [90, 91] and study patching rates [165]. This global view has also enabled a deeper and more precise understanding of recent cryptographic vulnerabilities such as Logjam [29] and DROWN [43] whose real-world impact is increased by widespread reuse of cryptographic parameters. This global view has actually enabled the discovery of new vulnerabilities like widespread random number generation failures that resulted in weak RSA and DSA keys [166, 205]. This dissertation presents new techniques for Internet scanning which further demonstrate the value of this approach as a means to improve the security of deployed cryptographic systems.

8.2. Summary of Impact

In this section, I attempt to summarize some of the measurable impact of this work. While certainly not a complete tabulation, the examples listed demonstrate the significant contributions this dissertation has made in securing cryptographic deployments.

8.2.1. Tools

Internet scanning. Throughout the course of my research, I developed several extensions to ZGrab, the application-layer scanner of the open source ZMap project [114]. These extensions allow for measuring invalid curve, twist, and small subgroup attacks against TLS, SSH, and IKE implementations, and for fine-grained testing of server supported cryptographic parameters. The IKE module for ZMap was built from scratch. All code is available at `github.com/eniac/zgrab`.

Factoring as a service. To demonstrate the insecurity of 512-bit RSA keys, my colleagues and I developed a tool to run a modified version of the CadoNFS [290] factoring software on a cloud computing platform. Our results showed that a 512-bit RSA key can be factored in as few as 4 hours at a cost of $75. Since releasing this tool, we have received reports that our tool has even been used to combat ransomware attacks! [1]

8.2.2. Protocols and standards

Recommendations against use of RFC 5114 DSA groups. RFC 5114 [206] published three finite field groups (Groups 22, 23, and 24) based on non-safe primes for use in Diffie-Hellman for IETF protocols. As described in Chapter 2, securely using non-safe primes requires implementations to include additional expensive validation checks; further, some protocols were designed to support only safe primes and did not build in mechanisms for implementations to properly validate Diffie-Hellman parameters when using non-safe primes. As a result of my work, as well as the work of Fried et al. [130] in pointing out that the provenance of the groups is not publicly verifiable, these groups have been recommended against by modern standards, including RFC 8247, which gives guidelines for

[1] from private communication

IKEv2 deployments [237]. [2]

TLSv1.3. My work has demonstrated several flaws in TLSv1.2 that have been subsequently addressed in the newest version of the protocol, TLSv1.3 [268]. Over the years, TLS has been shown to be vulnerable to a host of downgrade attacks such as Logjam (see Chapter 6) and CurveSwap (see Chapter 3). These attacks demonstrated the need for robust downgrade protection mechanisms in TLS, which were delivered for TLSv1.3 by Bhargavan et al. [69]. To further mitigate the threat of downgrade attacks, TLSv1.3 also removed support for legacy parameters that were included in TLSv1.2 for backwards-compatibility.

The attacks against TLS described in Chapter 2 and Chapter 6 partially exploit the ability of servers to choose their own custom Diffie-Hellman groups. TLSv1.3 removes this ability, and instead restricts Diffie-Hellman key exchange to a set of standarized groups that are believed to be secure.

In response to the myriad attacks against RSA key transport, including the DROWN attack described in Chapter 7, the TLS working group decided to remove support for RSA key exchange entirely in TLSv1.3.

8.2.3. Software changes

Diffie-Hellman validation and key reuse. The studies presented in this dissertation have revealed numerous weaknesses surrounding Diffie-Hellman validation and key reuse. Server-side finite field Diffie-Hellman (FFDH) implementations that were patched as a result of the study in Chapter 2 include OpenSSL, Amazon ELB, Unbound DNS, GnuTLS, LibTomCrypt, Exim mail server, and products from Cisco, VMWare, and Microsoft. JSON Web Encryption (JWE) implementations that were patches a result of the case study in Chapter 3 include Cisco node-jose, jose2go, Nimbus JOSE+JWT, and jose4j. This study also resulted in bug fixes for a multiplication error in NSS and Java cryptographic libraries.

SSLv2 support removed from libraries. Although SSLv2 had been deprecated for

[2] `https://mailarchive.ietf.org/arch/msg/saag/is67FiG6h1ApM6niKU6o-p8gkCo`

decades, support for the protocol was still present in several major cryptographic libraries at the time of discovery of the DROWN attack, including OpenSSL. Measurements from March 2016 showed that 33% of HTTPS servers were vulnerable at the time of the attack disclosure; however, SSL Labs estimates that only 1.2% of HTTPS servers are vulnerable as of 2019. [3]

Libgcrypt validation for X25519. After the disclosure of the side-channel attack discussed in Chapter 4, Libgcrypt implemented the countermeasure of rejecting low-order inputs. This countermeasure does not address the underlying issue that Libgcrypt uses non-constant time arithmetic operations, but it does mitigate the discovered attacks.

Browsers. Major browsers raised the minimum acceptable ephemeral Diffie-Hellman key size to 1024 bits in the immediate aftermath of the Logjam attack (see Chapter 6). [4] [5] [6] While computing a 1024-bit discrete log is still expected to be within the reach of nation-state adversaries, this change helps to mitigate the immediate risk from more resource-constrained adversaries.

[3] https://dev.ssllabs.com/ssl-pulse

[4] https://groups.google.com/a/chromium.org/forum/%23!topic/security-dev/WyGIpevBV1s

[5] https://blog.mozilla.org/security/2015/07/02/mitigating-logjam-enforcing-stronger-diffie-hellman-key-exchange/

[6] https://support.apple.com/en-us/HT205020

BIBLIOGRAPHY

[1] FY 2013 congressional budget justification. Media leak. http://cryptome.org/2013/08/spy-budget-fy13.pdf.

[2] Censys. https://censys.io/.

[3] FreeS/WAN. http://www.freeswan.org/.

[4] Hashcat. http://hashcat.net.

[5] Libreswan. https://libreswan.org/.

[6] Innov8 experiment profile. Media leak, . http://www.spiegel.de/media/media-35509.pdf.

[7] TURMOIL/APEX/APEX high level description document. Media leak, . http://www.spiegel.de/media/media-35513.pdf.

[8] GALLANTWAVE@scale. Media leak, . http://www.spiegel.de/media/media-35514.pdf.

[9] VALIANTSURF (VS): Capability levels. Media leak, . http://www.spiegel.de/media/media-35517.pdf.

[10] POISONNUT – WikiInfo. Media leak, . http://www.spiegel.de/media/media-35519.pdf.

[11] VPN SigDev basics. Media leak, . http://www.spiegel.de/media/media-35520.pdf.

[12] SPIN 15 VPN story. Media leak, . http://www.spiegel.de/media/media-35522.pdf.

[13] VALIANTSURF – WikiInfo. Media leak, . http://www.spiegel.de/media/media-35527.pdf.

[14] Fielded capability: End-to-end VPN SPIN 9 design review. Media leak, . http://www.spiegel.de/media/media-35529.pdf.

[15] LONGHAUL – WikiInfo. Media leak, . http://www.spiegel.de/media/media-35533.pdf.

[16] What your mother never told you about SIGDEV analysis. Media leak, . http://www.spiegel.de/media/media-35551.pdf.

[17] Openswan. https://www.openswan.org/.

[18] Pidgin. https://www.pidgin.im/.

[19] SIGINT strategy. Media leak. http://www.nytimes.com/interactive/2013/11/23/us/politics/23nsa-sigint-strategy-document.html.

[20] TURMOIL VPN processing. Media leak, Oct. 2009. http://www.spiegel.de/media/media-35526.pdf.

[21] TURMOIL IPsec VPN sessionization. Media leak, Aug. 2009. http://www.spiegel.de/media/media-35528.pdf.

[22] APEX active/passive exfiltration. Media leak, Aug. 2009. http://www.spiegel.de/media/media-35671.pdf.

[23] Intro to the VPN exploitation process. Media leak, Sept. 2010. http://www.spiegel.de/media/media-35515.pdf.

[24] stud: The scalable TLS unwrapping daemon, 2012. https://github.com/bumptech/stud/blob/19a7f19686bcdbd689c6fbea31f68a276e62d886/stud.c#L593.

[25] CVE-2015-0293. Available from MITRE, CVE-ID CVE-2015-0293., 2015. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0293.

[26] CVE-2015-3240. Available from MITRE, CVE-ID CVE-2015-3240., Aug. 2015. http://cve.mitre.org/cgi-bin/cvename.cgi?name=2015-3240.

[27] CVE-2016-0701. Available from MITRE, CVE-ID CVE-2016-0701., Jan. 2016. http://cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0701.

[28] O. Acıiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *CHES*, pages 110–124, 2010.

[29] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security (CCS '15)*, 2015.

[30] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A. D. Keromytis, and O. Reingold. Just fast keying: Key agreement in a hostile internet. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):242–273, 2004.

[31] T. Akishita and T. Takagi. Zero-value point attacks on elliptic curve cryptosystem. In *ISC 2003*, pages 218–233, 2003.

[32] N. J. AlFardan and K. G. Paterson. Lucky thirteen: breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[33] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt. On the security of RC4 in TLS. In *USENIX Security Symposium*, pages 305–320, 2013.

[34] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom. Amplifying side channels through performance degradation. In *ACSAC*, Los Angeles, CA, US, Dec. 2016.

[35] E. Allman, J. Callas, M. Delany, M. Libbey, J. Fenton, and M. Thomas. DomainKeys identified mail (DKIM) signatures, 2007.

[36] Amazon. Amazon Elastic Load Balancer, . `https://aws.amazon.com/elasticloadbalancing/`.

[37] Amazon. Amazon EC2, . `https://aws.amazon.com/ec2/`.

[38] R. Anderson and S. Vaudenay. Minding your p's and q's. In *Proceedings of ASIACRYPT*, 1996.

[39] A. Antipa, D. Brown, A. Menezes, R. Struik, and S. Vanstone. Validation of elliptic curve public keys. In *International Workshop on Public Key Cryptography*, pages 211–223. Springer, 2003.

[40] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, Internet Society, March 2005.

[41] D. Auble, M. Jette, et al. Slurm documentation. `http://slurm.schedmd.com/`. Accessed: 2015-09-19.

[42] J. P. Aumasson. Should curve25519 keys be validated? `https://research.kudelskisecurity.com/2017/04/25/should-ecdh-keys-be-validated/`, Apr. 2017.

[43] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt. DROWN: Breaking TLS with SSLv2. In *25th USENIX Security Symposium*, Aug. 2016.

[44] A. Ayer. git-crypt — transpartent file encryption in git. `https://www.agwa.name/projects/git-crypt/`.

[45] S. Bai, C. Bouvier, A. Filbois, P. Gaudry, L. Imbert, A. Kruppa, F. Morain, E. Thomé, and P. Zimmermann. `cado-nfs`, an implementation of the number field sieve algorithm, 2014. Release 2.1.1.

[46] R. Barbulescu. *Algorithmes de logarithmes discrets dans les corps finis*. PhD thesis, Université de Lorraine, France, 2013.

[47] R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *Eurocrypt*, 2014.

[48] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology–CRYPTO 2012*, pages 608–625. Springer, 2012.

[49] E. Barker, L. Chen, A. Roginsky, and M. Smid. NIST SP 800-56A: Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revision 2), 2013.

[50] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. NIST SP 800-57: Recommendation for key management (revision 4), 2016.

[51] P. Belgarric, P.-A. Fouque, G. Macario-Rat, and M. Tibouchi. Side-channel analysis of Weierstrass and Koblitz curve ECDSA on Android smartphones. In *CT-RSA 2016*, pages 236–252. Springer, 2016.

[52] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES 2014*, pages 75–92, 2014.

[53] D. J. Bernstein. How to find smooth parts of integers. `http://cr.yp.to/factorization/smoothparts-20040510.pdf`, 2004.

[54] D. J. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/papers.html#cachetiming`, 2005.

[55] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *PKC*, pages 207–228, New-York, NY, US, Apr. 2006.

[56] D. J. Bernstein. A state-of-the-art Diffie-Hellman function. `https://cr.yp.to/ecdh.html`, 2006.

[57] D. J. Bernstein and T. Lange. Batch NFS. In *Selected Areas in Cryptography*, 2014.

[58] D. J. Bernstein and T. Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography, Jan. 2014. `https://safecurves.cr.yp.to/`.

[59] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *LatinCrypt'12*, pages 159–176, Santiago, CL, Oct. 2012.

[60] D. J. Bernstein, C. Chuengsatiansup, and T. Lange. Curve41417: Karatsuba revisited. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 316–334. Springer, 2014.

[61] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming

the composite state machines of TLS. In *IEEE Symposium on Security and Privacy*, 2015.

[62] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. FREAK: Factoring RSA export keys. `https://www.smacktls.com/#freak`, 2015.

[63] K. Bhargavan and G. Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH. In *Network and Distributed System Security Symposium*, 2016.

[64] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing tls with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459. IEEE, 2013.

[65] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security & Privacy*. IEEE, 2014.

[66] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. Proving the tls handshake secure (as it is). In *International Cryptology Conference*, pages 235–255. Springer, 2014.

[67] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti. Verified contributive channel bindings for compound authentication. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[68] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. IETF RFC 7627, Sept. 2015.

[69] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin. Downgrade resilience in key-exchange protocols. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 506–525. IEEE, 2016.

[70] I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In *Annual International Cryptology Conference*, pages 131–146. Springer, 2000.

[71] O. Billet and M. Joye. The Jacobi model of an elliptic curve and side-channel analysis. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC) 2015*, pages 34–42. Springer, 2003.

[72] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). IETF RFC 4492, May 2006.

[73] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. In *CRYPTO*, pages 1–12, 1998.

[74] J. W. Bos, J. A. Halderman, N. Heninger, J. Moore, M. Naehrig, and E. Wustrow. Elliptic curve cryptography in practice. In *International Conference on Financial Cryptography and Data Security*, pages 157–175. Springer, 2014.

[75] C. Bouvier, P. Gaudry, L. Imbert, H. Jeljeli, and E. Thomé. New record for discrete logarithm in a prime finite field of 180 decimal digits, 2014. `http://caramel.loria.fr/p180.txt`.

[76] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. IETF RFC 2119, Mar. 1997.

[77] W. Breyha, D. Durvaux, T. Dussa, L. A. Kaplan, F. Mendel, C. Mock, M. Koschuch, A. Kriegisch, U. Pöschl, R. Sabet, B. San, R. Schlatterbeck, T. Schreck, A. Würstlein, A. Zauner, and P. Zawodsky. Better crypto – applied crypto hardening, 2016. Available at `https://bettercrypto.org/static/applied-crypto-hardening.pdf`.

[78] É. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography: 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002 Paris, France, February 12–14, 2002 Proceedings*, pages 335–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45664-3.

[79] D. R. L. Brown. SEC 1: Elliptic curve cryptography. *Certicom Research*, 2009.

[80] D. R. L. Brown. SEC 2: Recommended elliptic curve domain parameters. *Certicom Research*, 2010.

[81] M. Brown, D. Hankerson, J. Lopez, and A. Menezes. Software implementation of the nist elliptic curves over prime fields. In *TOPICS IN CRYPTOLOGY – CT-RSA 2001, VOLUME 2020 OF LNCS*, pages 250–265. Springer, 2001.

[82] B. B. Brumley and R. M. Hakala. Cache-timing template attacks. In *ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 667–684. Springer, 2009.

[83] B. B. Brumley and N. Tuveri. Remote timing attacks are still practical. In *ESORICS 2011*, pages 355–371. Springer, 2011.

[84] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[85] Bureau of Industry and Security. Export administration regulations. `http://www.bis.doc.gov/index.php/regulations/export-administration-regulations-ear`, 2015.

[86] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP message format. RFC 4880, Nov. 2007.

[87] R. Canetti and H. Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In *Crypto*, 2002.

[88] S. Cavallar, B. Dodson, A. K. Lenstra, W. Lioen, P. L. Montgomery, B. Murphy, H. te Riele, K. Aardal, J. Gilchrist, G. Guillerm, P. Leyland, J. Marchand, F. Morain, A. Muffett, C. Putnam, C. Putnam, and P. Zimmermann. Factorization of a 512-bit RSA modulus. In B. Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67517-4.

[89] W.-T. Chang and A. Langley. QUIC crypto, 2014. `https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit?pli=1`.

[90] S. Checkoway, M. Fredrikson, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. On the practical exploitability of Dual EC in TLS implementations. In *21nd ACM Conference on Computer and Communications Security*, Nov. 2014.

[91] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham. A systematic analysis of the Juniper Dual EC incident. In *23nd ACM Conference on Computer and Communications Security*, Oct. 2016.

[92] G. Childers. NFS@home. `http://escatter11.fullerton.edu/nfs/`.

[93] D. V. Chudnovsky and G. V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7(4):385–434, 1986.

[94] M. Ciet and M. Joye. (Virtually) free randomization techniques for elliptic curve cryptography. In *ICICS 2003*, pages 348–359. Springer, 2003.

[95] Cisco. Security for VPNs with IPsec configuration guide, 2016. `http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/sec_conn_vpnips/configuration/xe-3s/sec-sec-for-vpns-w-ipsec-xe-3s-book.html`.

[96] A. Commeine and I. Semaev. An algorithm to solve the discrete logarithm problem with the number field sieve. In *PKC*, 2006.

[97] D. Coppersmith. Solving linear equations over GF(2) via block Wiedemann algorithm. *Math. Comp.*, 62(205), 1994.

[98] D. Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, 1994.

[99] J. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *CHES 1999*, pages 292–302, 1999.

[100] B. Cox. Auditing GitHub users SSH key quality, June 2015.

[101] R. Crandall and C. B. Pomerance. *Prime Numbers: A Computational Perspective*, volume 182. Springer, 2006.

[102] C. Cremers. Key exchange in ipsec revisited: Formal analysis of ikev1 and ikev2. *Computer Security–ESORICS 2011*, pages 315–334, 2011.

[103] J. de Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium*, Washington, D.C., Aug. 2015. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter`.

[104] M. DeHaan. Ansible. `http://www.ansible.com`.

[105] B. den Boer. Diffie-Hellman is as strong as discrete log for certain primes. In *Crypto*, 1988.

[106] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[107] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446, 2018.

[108] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[109] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1197–1210. ACM, 2015.

[110] N. Drucker and S. Gueron. Selfie: reflections on TLS 1.3 with PSK. Cryptology ePrint Archive, Report 2019/347, 2019. `https://eprint.iacr.org/2019/347`.

[111] T. Duong. Why not validate Curve25519 public keys could be harmful, Sept. 2015.

[112] T. Duong and J. Rizzo. Here come the XOR ninjas. *White paper, Netifera*, 2011.

[113] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS certificate ecosystem. In *Proceedings of the 13th Internet Measurement Conference*, Oct. 2013.

[114] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Security Symposium*, Aug. 2013.

[115] Z. Durumeric, M. Bailey, and J. A. Halderman. An Internet-wide view of Internet-wide scanning. In *Proceedings of the 23rd USENIX Security Symposium*, Aug. 2014.

[116] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Proceedings of the 2014 Internet Measurement Conference*, Nov. 2014.

[117] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by Internet-wide scanning. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[118] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither snow nor rain nor MITM... an empirical analysis of email delivery security. In *Proceedings of Internet Measurement Conference (IMC) 2015*, 2015.

[119] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, K. Thomas, V. Eranti, N. Lidzborski, E. Bursztein, M. Bailey, and J. A. Halderman. The Matter of Heartbleed. In *Proceedings of the 15th ACM Internet Measurement Conference*, Oct. 2015.

[120] M. Elkins, D. D. Torto, R. Levien, and T. Roessler. MIME security with OpenPGP. RFC 3156, 2001.

[121] Y. Eric. SSLeay, 1995. `ftp://ftp.pl.vim.org/vol/rzm1/replay.old/libraries/SSL.eay/SSLeay-0.5.1a.tar.gz`.

[122] Exim. Exim Internet mailer. `http://www.exim.org/`.

[123] J. Fan and I. Verbauwhede. An updated survey on secure ECC implementations: Attacks, countermeasures and cost. In *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*, pages 265–282, 2012.

[124] J. Fan, X. Guo, E. D. Mulder, P. Schaumont, B. Preneel, and I. Verbauwhede. State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures. In *HOST 2010*, pages 76–87, 2010.

[125] J. Fan, B. Gierlichs, and F. Vercauteren. To infinity and beyond: Combined attack on ECC using points of low order. In *Cryptographic Hardware and Embedded Systems CHES 2011*, pages 143–159. Springer, 2011.

[126] N. Ferguson and B. Schneier. A cryptographic evaluation of IPsec. *Counterpane Internet Security, Inc*, 3031, 2000.

[127] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In S. Vaudenay and A. M. Youssef, editors, *Selected Areas in Cryptography: 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16–17, 2001 Revised Papers*, pages 1–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-45537-0. doi: 10.1007/3-540-45537-X_1. URL `http://dx.doi.org/10.1007/3-540-45537-X_1`.

[128] P. A. Fouque, R. Lercier, D. Réal, and F. Valette. Fault attack on elliptic curve montgomery ladder implementation. In *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 92–98, Aug 2008.

[129] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (SSL) protocol version 3.0. RFC 6101, 2011.

[130] J. Fried, P. Gaudry, N. Heninger, and E. Thomé. A kilobit hidden snfs discrete logarithm computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 202–231. Springer, 2017.

[131] M. Friedl, N. Provos, and W. Simpson. Diffie-Hellman group exchange for the Secure Shell (SSH) transport layer protocol. IETF RFC 4419, 2006.

[132] D. Fu and J. Solinas. Elliptic curve groups modulo a prime (ecp groups) for ike and ikev2. IETF RFC 5903, 2010.

[133] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[134] S. D. Galbraith and P. Gaudry. Recent progress on the elliptic curve discrete logarithm problem. *Des. Codes Cryptography*, 78(1):51–72, Jan. 2016. ISSN 0925-1022.

[135] S. Gallagher. Google dumps plans for OpenSSL in Chrome, takes own Boring road, July 2014. `http://arstechnica.com/information-technology/2014/07/google-dumps-plans-for-openssl-in-chrome-takes-own-boring-road/`.

[136] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, -, 2016.

[137] W. Geiselmann and R. Steinwandt. Non-wafer-scale sieving hardware for the NFS: Another attempt to cope with 1024-bit. In *Eurocrypt*, 2007.

[138] W. Geiselmann, H. Kopfer, R. Steinwandt, and E. Tromer. Improved routing-based linear algebra for the number field sieve. In *Information Technology: Coding and Computing*, 2005.

[139] D. Genkin, I. Pipman, and E. Tromer. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. In *CHES 2014*, pages 242–260. Springer, 2014. Extended version: Cryptology ePrint Archive, Report 2014/626.

[140] D. Genkin, A. Shamir, and E. Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014*, pages 444–461 (vol. 1). Springer, 2014.

[141] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *CHES 2015*, pages 207–228, 2015. Extended version: Cryptology ePrint Archive, Report 2015/170.

[142] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer. Physical key extraction attacks on PCs. *Commun. ACM*, 59(6):70–79, 2016.

[143] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer. ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs. In *CT-RSA 2016*, pages 219–235, 2016.

[144] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *ACM Conference on Computer and Communications Security CCS 2016*, pages 1626–1638, Oct. 2016.

[145] D. Genkin, L. Valenta, and Y. Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 845–858. ACM, 2017.

[146] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[147] D. Gillmor. Negotiated finite field Diffie-Hellman ephemeral parameters for TLS. IETF Internet Draft, May 2015.

[148] D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). IETF RFC 7919, Aug. 2016.

[149] T. Glaser. OpenPGP plugin for Pidgin.

[150] GMP-ECM Development Team. GMP-ECM, an implementation of the elliptic curve method for integer factorization, 2016. `http://ecm.gforge.inria.fr/`.

[151] GnuPG. GnuPG Frontends. `https://www.gnupg.org/related_software/frontends.html`, .

[152] GnuPG. GNU Privacy Guard. `https://www.gnupg.org`, .

[153] D. M. Gordon. Designing and detecting trapdoors for discrete log cryptosystems. In *Crypto*, 1992.

[154] D. M. Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM Journal of Discrete Math*, 1993.

[155] L. Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In *PKC 2003*, pages 199–210, 2003.

[156] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, Gauss, and reload — a cache attack on the BLISS lattice-based signature scheme. In *CHES*, pages 323–345, Aug. 2016.

[157] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, pages 897–912, Washington, DC, US, Aug. 2015.

[158] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *IEEE S&P*, pages 490–505, Oakland, CA, US, May 2011.

[159] P. Gutmann. Cryptlib, kg_dlp.c, 2010. `http://www.cypherpunks.to/~peter/cl343_beta.zip`.

[160] O. H., P. S. Dev., H. P., and V. Consortium. Determining strengths for public keys used for exchanging symmetric keys. IETF RFC 3766, Apr. 2004.

[161] M. Hamburg. Ed448-Goldilocks, a new elliptic curve. *IACR Cryptology ePrint Archive*, 2015:625, 2015.

[162] R. Hamilton. QUIC discovery, Oct. 2010. `https://docs.google.com/document/d/1i4m7DbrWGgXafHxwl8SwIusY2ELUe8WX258xt2LFxPM/edit#`.

[163] D. Harkins. Brainpool elliptic curves for the internet key exchange (ike) group description registry. IETF RFC 6932, 2013.

[164] D. Harkins and D. Carrel. The Internet key exchange (IKE). IETF RFC 2409, Nov. 1998.

[165] M. Hastings, J. Fried, and N. Heninger. Weak keys remain widespread in network devices. In *Proceedings of the 2016 Internet Measurement Conference*, pages 49–63. ACM, 2016.

[166] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[167] K. Hickman. The SSL protocol. `https://tools.ietf.org/html/draft-hickman-netscape-ssl-00`, Feb. 1995.

[168] C. Hlauschek, M. Gruber, F. Fankhauser, and C. Schanes. Prying open Pandora's box: KCI attacks against TLS. In *9th USENIX Workshop on Offensive Technologies (WOOT '15)*, Aug. 2015.

[169] R. Holz, J. Amann, O. Mehani, M. Wachs, and M. A. Kaafar. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. In S. Capkun, editor, *Network and Distributed System Security SymposiumNDSS 2016,*, Geneva, Switzerland, Feb. 2016. Internet Society, Internet Society. ISBN 1-891562-41-X. doi: 10.14722/ndss.2016.23055.

[170] R. Housley. Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. IETF RFC 7696, 2015.

[171] E. Hughes. How to give a math lecture at a party, 2000.

[172] IANA. Transport Layer Security (TLS) Parameters. `http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`, May 2017.

[173] IANIX. Things that use Curve25519. `https://ianix.com/pub/curve25519-deployment.html`, Mar. 2017.

[174] M. S. İnci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Cache attacks enable bulk key recovery on the cloud. In *CHES*, pages 368–388, 2016.

[175] International Computer Science Institute. The ICSI certificate notary.

[176] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! a fast, cross-VM attack on AES. In *RAID*, pages 299–319, Gothenburg, Sweden, Sept. 2014.

[177] T. Jager, S. Schinzel, and J. Somorovsky. Bleichenbacher's attack strikes again: Breaking pkcs#1 v1.5 in xml encryption. In S. Foresti, M. Yung, and F. Martinelli, editors, *Computer Security – ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 752–769. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-33167-1. doi: 10.1007/978-3-642-33167-1_43. URL `http://dx.doi.org/10.1007/978-3-642-33167-1_43`.

[178] T. Jager, K. G. Paterson, and J. Somorovsky. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *NDSS*, 2013.

[179] T. Jager, J. Schwenk, and J. Somorovsky. Practical invalid curve attacks on TLS-ECDH. In *Proceedings of the 20th European Symposium on Research in Computer Security*, 2015.

[180] T. Jager, J. Schwenk, and J. Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1185–1196, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813657. URL `http://doi.acm.org/10.1145/2810103.2813657`.

[181] W. Jeffrey. Crypto++, Nov. 2015. `https://github.com/weidai11/cryptopp/blob/48809d4e85c125814425c621d8d0d89f95405924/nbtheory.cpp#L1029`.

[182] A. Jivsov. Elliptic curve cryptography (ECC) in OpenPGP. RFC 6637, June 2012.

[183] A. Joux and R. Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the Gaussian integer method. *Math. Comp.*, 72(242):953–967, 2003.

[184] M. Joye and S. Yen. The Montgomery powering ladder. In *Cryptographic Hardware and Embedded Systems (CHES) 2002*, pages 291–302. Springer, 2002.

[185] Juniper TechLibrary. VPN feature guide for security devices, 2016. `http://www.juniper.net/documentation/en_US/junos15.1x49/topics/reference/configuration-statement/security-edit-dh-group.html`.

[186] B. Kaliski. PKCS #1: RSA Encryption Version 1.5. IETF RFC 2313 (Informational), Mar. 1998. Obsoleted by RFC 2437.

[187] E. Käsper. Fix reachable assert in SSLv2 servers. OpenSSL patch, Mar. 2015. `https://github.com/openssl/openssl/commit/86f8fb0e344d62454f8daf3e15236b2b59210756`.

[188] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet Key Exchange protocol version 2 (IKEv2). IETF RFC 7296, Oct. 2014.

[189] C. Kaufman et al. Internet Key Exchange (IKEv2) protocol. IETF RFC 4306, Dec. 2005.

[190] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *CANS'16*, pages 573–582, Nov. 2016.

[191] S. Kent. IP authentication header. RFC 4302, Dec. 2005.

[192] S. Kent. IP encapsulating security payload (ESP). RFC 4303, Dec. 2005.

[193] T. Kivinen and M. Kojo. More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). IETF RFC 3526, May 2003.

[194] T. Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers, 2006. `http://www.hyperelliptic.org/tanja/SHARCS/talks06/thorsten.pdf`.

[195] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *Crypto*, 2010.

[196] T. Kleinjung, A. K. Lenstra, D. Page, and N. P. Smart. Using the cloud to determine key strengths. In *Progress in Cryptology-INDOCRYPT 2012*, pages 17–39, 2012.

[197] V. Klima, O. Pokorny, and T. Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, pages 426–440, 2003.

[198] D. E. Knuth. The art of computer programming. *Seminumeral Algorithms*, 2:257–258, 1981.

[199] O. M. Kolkman, W. M. Mekking, and R. M. Gieben. DNSSEC Operational Practices, Version 2. RFC 6781, Internet Society, December 2012.

[200] A. Langley and M. Hamburg. Elliptic curves for security. IETF RFC 7748, Jan. 2016.

[201] A. Langley, N. Modaduga, and B. Moeller. Transport Layer Security (TLS) False Start. IETF RFC 7918, Aug. 2016.

[202] A. K. Lenstra and H. W. Lenstra, Jr., editors. *The Development of the Number Field Sieve*. Springer, 1993.

[203] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982. ISSN 0025-5831. URL `http://dx.doi.org/10.1007/BF01457454`. 10.1007/BF01457454.

[204] A. K. Lenstra, H. W. Lenstra Jr, M. S. Manasse, and J. M. Pollard. *The number field sieve*. Springer, 1993.

[205] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Public keys. In *32nd International Cryptology Conference*, Aug. 2012.

[206] M. Lepinski and S. Kent. Additional Diffie-Hellman Groups for Use with IETF Standards. IETF RFC 5114, 2008.

[207] C. H. Lim and P. J. Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *Advances in Cryptology–CRYPTO'97*, volume 1294 of *LNCS*, pages 249–263, 1997.

[208] M. Lipacis. Semiconductors: Moore stress = structural industry shift. Technical report, Jefferies, 2012.

[209] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy 2015*. IEEE, 2015.

[210] J. Lloyd. Botan, 2016. `https://github.com/randombit/botan`.

[211] D. Martin. Secure protocols in a hostile world. Bristol Cryptography Blog, Sept. 2015. `http://bristolcrypto.blogspot.co.il/2015/09/secure-protocols-in-hostile-world.html`.

[212] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet security association and key management protocol ISAKMP. IETF RFC 2408, Nov. 1998.

[213] U. M. Maurer. Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms. In *Crypto*, 1994.

[214] U. M. Maurer and S. Wolf. Diffie-Hellman oracles. In *Crypto*, 1996.

[215] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, pages 62–72, 2012.

[216] W. Mayer, A. Zauner, M. Schmiedecker, and M. Huber. No need for black chambers: Testing TLS in the e-mail ecosystem at large. *CoRR*, abs/1510.08646, 2015. URL `http://arxiv.org/abs/1510.08646`.

[217] C. Meadows. Analysis of the Internet key exchange protocol using the NRL protocol analyzer. In *IEEE Symposium on Security and Privacy*, 1999.

[218] A. Menezes and B. Ustaoglu. On reusing ephemeral keys in diffie-hellman key agreement protocols. *International Journal of Applied Cryptography*, 2(2):154–158, 2010.

[219] J. Merkle and M. Lochter. Elliptic curve cryptography (ECC) Brainpool standard curves and curve generation. IETF RFC 5639, Mar. 2010.

[220] C. Meyer and J. Schwenk. Lessons learned from previous SSL/TLS attacks – A brief chronology of attacks and weaknesses. IACR Cryptology ePrint Archive, Report 2013/049, 2013.

[221] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium*, pages 733–748, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer`.

[222] Microsoft Security Bulletin MS15-055. Vulnerability in Schannel could allow information disclosure, May 2015.

[223] Microsoft Windows Networking Team. VPN interoperability guide for Windows Server 2012 R2, 2014. `https://blogs.technet.microsoft.com/networking/2014/12/26/vpn-interoperability-guide-for-windows-server-2012-r2/`.

[224] MikroTik. MikroTik Manual:IP/IPsec, Feb. 2018. `https://wiki.mikrotik.com/wiki/Manual:IP/IPsec`.

[225] B. Möller, T. Duong, and K. Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[226] C. Monico. GGNFS. `http://www.math.ttu.edu/~cmonico/software/ggnfs/`.

[227] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.

[228] P. L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In L. C. Guillou and J.-J. Quisquater, editors, *Advances in Cryptology–EUROCRYPT '95*, pages 106–120, 1995. ISBN 978-3-540-59409-3.

[229] Mozilla. NSS dh.c. `https://hg.mozilla.org/projects/nss/file/tip/lib/freebl/dh.c`.

[230] Mozilla. Mozilla bug tracker, Nov. 2015. `https://bugzilla.mozilla.org/show_bug.cgi?id=1160139`.

[231] Mozilla. Overview of NSS, Sept. 2015. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Overview`.

[232] Mozilla. Mozilla foundation security advisory [8th Aug 2017], Aug. 2017. `https://www.mozilla.org/en-US/security/advisories/mfsa2017-18/#CVE-2017-7781`.

[233] National Institute of Standards and Technology. FIPS 140-2: Security requirements for cryptographic modules, may 2001.

[234] Nguyen and Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, 2002. ISSN 1432-1378.

[235] P. Q. Nguyen. Can we trust cryptographic software? cryptographic flaws in gnu privacy guard v1. 2.3. In *EUROCRYPT*, volume 4, pages 555–570. Springer, 2004.

[236] Q. Nguyen. Practical Cryptanalysis of JSON Web Token and Galois Counter Mode's Implementations. In *Real World Crypto Conference 2017*, 2017. http://www.realworldcrypto.com/rwc2017.

[237] Y. Nir, T. Kivinen, P. Wouters, and D. Migualt. Algorithm Implementation Requirements and Usage Guidance for the Internet Key Exchange Protocol Version 2 (IKEv2). IETF RFC 8247, Sept. 2017.

[238] NIST. Finite field cryptography based samples. `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/KS_FFC_All.pdf`.

[239] NIST. FIPS PUB 186-4: Digital signature standard (DSS), 2013.

[240] Oak Ridge National Laboratory. Introducing Titan, 2012. `https://www.olcf.ornl.gov/titan`.

[241] A. M. Odlyzko. The future of integer factorization, July 1995. `http://www.dtc.umn.edu/~odlyzko/doc/future.of.factoring.pdf`.

[242] D. of State. International Traffic in Arms Regulations. https://epic.org/crypto/export_controls/itar.html, Apr. 1992.

[243] K. Okeya, H. Kurumatani, and K. Sakurai. Elliptic curves with the Montgomery-form and their cryptographic applications. In *Public Key Cryptography (PKC) 2000*, pages 238–257. Springer, 2000.

[244] OpenBSD. OpenBSD iked.conf, Jan. 2018. https://man.openbsd.org/iked.conf.5.

[245] OpenSSL. Change log. https://www.openssl.org/news/changelog.html#x0.

[246] OpenSSL. OpenSSL changes, Jan. 2015. https://www.openssl.org/news/cl102.txt.

[247] OpenSSL. OpenSSL security advisory [28th Jan 2016], Jan. 2016. https://www.openssl.org/news/secadv/20160128.txt.

[248] Oracle. Oracle critical patch update advisory [18th Jul 2017], July 2017. http://www.oracle.com/technetwork/security-advisory/cpujul2017-3236622.html.

[249] H. Orman. The OAKLEY Key Determination Protocol. IETF RFC 2412, 1998.

[250] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA 2006*, pages 1–20. Springer, 2006.

[251] J. Papadopoulos. Msieve. http://www.boo.net/~jasonp/qs.html.

[252] K. G. Paterson, B. Poettering, and J. C. Schuldt. Big bias hunting in Amazonia: Large-scale computation and exploitation of RC4 biases (invited paper). In P. Sarkar and T. Iwata, editors, *Advances in Cryptology—ASIACRYPT 2014*, pages 398–419, 2014.

[253] P. Pennock. Exim TLS Security, DH and standard parameters, Oct. 2016. https://lists.exim.org/lurker/message/20161008.231103.c70b2da8.en.html.

[254] C. Percival. Cache missing for fun and profit. Presented at BSDCan. http://www.daemonology.net/hyperthreading-considered-harmful, 2005.

[255] C. Pereida García and B. B. Brumley. Constant-Time Callees with Variable-Time Callers. In *USENIX Security Symposium 2017*, pages 83–98, Sept. 2017.

[256] C. Pereida García, B. B. Brumley, and Y. Yarom. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *CCS'16*, pages 1639–1650, Oct. 2016.

[257] T. Perrin. X25519 and zero outputs, May 2017.

[258] D. Piper. The Internet IP security domain of interpretation for ISAKMP. IETF RFC 2407, Nov. 1998.

[259] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24(1), 1978.

[260] W. Polk, R. Housley, and L. Bassham. Algorithms and identifiers for the internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. IETF RFC 3279, Apr. 2002.

[261] J. M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[262] M. J. Pollard. Kangaroos, Monopoly and discrete logarithms. *Journal of Cryptology*, 2000.

[263] J. Van de Pol, N. P. Smart, and Y. Yarom. Just a little bit more. In *CT-RSA 2015*, pages 3–21, 2015.

[264] C. Pomerance. A tale of two sieves. In *Notices Amer. Math. Soc*, 1996.

[265] B. C. Project. Browser capabilities project, Aug. 2017.

[266] T. E. Project. Enigmail: A simple interface for OpenPGP email security.

[267] R. Ransom. Leading zero bits in the Montgomery ladder. IETF mailing list, July 2014. `https://www.ietf.org/mail-archive/web/cfrg/current/msg04749.html`.

[268] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. IETF RFC 8446, 2018.

[269] I. Ristić. `https://www.trustworthyinternet.org/ssl-pulse/`.

[270] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[271] J. Rizzo and T. Duong. The CRIME attack. EKOparty Security Conference, 2012.

[272] J. Roskind. QUIC design document, 2013. `https://docs.google.com/a/chromium.org/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34`.

[273] A. Sanso. OpenSSL Key Recovery Attack on DH small subgroups, Jan. 2016. http://blog.intothesymmetry.com/2016/01/openssl-key-recovery-attack-on-dh-small.html.

[274] A. Sanso. Critical vulnerability in JSON Web Encryption (JWE) - RFC 7516 ,

Mar. 2017. http://blog.intothesymmetry.com/2017/03/critical-vulnerability-in-json-web.html.

[275] A. Sanso. CVE-2017-7781/CVE-2017-10176: Issue with elliptic curve addition in mixed Jacobian-affine coordinates in Firefox/Java, Aug. 2017. http://blog.intothesymmetry.com/2017/08/cve-2017-7781cve-2017-10176-issue-with.html.

[276] O. Schirokauer. Virtual logarithms. *J. Algorithms*, 57(2):140–147, 2005.

[277] I. Semaev. Special prime numbers and discrete logs in finite prime fields. *Math. Comp.*, 71(237):363–377, 2002.

[278] I. Semaev. New algorithm for the discrete logarithm problem on elliptic curves. Cryptology ePrint Archive, Report 2015/310, 2015. `http://eprint.iacr.org/2015/310`.

[279] D. Shanks. Class number, a theory of factorization, and genera. In *Proceedings of Symposia in Pure Math*, volume 20, 1969.

[280] Y. Sheffer and S. Fluhrer. Additional Diffie-Hellman tests for the Internet Key Exchange protocol version 2 (IKEv2). IETF RFC 6989, 2013.

[281] D. Smith. All TI signing keys factored, September 2009.

[282] Spiegel Staff. Prying eyes: Inside the NSA's war on Internet security. Der Spiegel, Dec 2014. `http://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html`.

[283] D. Springall, Z. Durumeric, and J. A. Halderman. Measuring the security harm of tls crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 33–47, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4526-2.

[284] D. Stebila and J. Green. Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer. IETF RFC 5656, 2009.

[285] A. Steffen. strongSwan. `https://strongswan.org`.

[286] W. Stein et al. Sage Mathematics Software (Version 6.5), 2015. `http://www.sagemath.org`.

[287] J. Steube. Optimizing computation of hash-algorithms as an attacker. In *Passwords*, Las Vegas, 2013. `http://hashcat.net/events/p13/js-ocohaaaa.pdf`.

[288] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In S. Halevi, editor, *Advances in Cryptology - CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 55–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN

978-3-642-03356-8. doi: 10.1007/978-3-642-03356-8_4. URL http://dx.doi.org/10.1007/978-3-642-03356-8_4.

[289] N. Sullivan. goto fail; exploring two decades of transport layer security. https://media.ccc.de/v/32c3-7438-goto_fail, Dec. 2015.

[290] T. C. D. Team. CADO-NFS, an implementation of the number field sieve algorithm. http://cado-nfs.gforge.inria.fr/, 2015.

[291] E. Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *J. Symbolic Comput.*, 33(5):757–775, 2002.

[292] Y. Tsunoo. Cryptanalysis of block ciphers implemented on computers with cache. *preproceedings of ISITA 2002*, 2002.

[293] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES*, volume 2779, pages 62–76. Springer, 2003.

[294] S. Turner and T. Polk. Prohibiting secure sockets layer (SSL) version 2.0. IETF RFC 6176 (Informational), Apr. 2011.

[295] L. Valenta. Bug 1837 - small subgroup attack, May 2016. https://bugs.exim.org/show_bug.cgi?id=1837.

[296] L. Valenta, S. Cohney, A. Liao, J. Fried, S. Bodduluri, and N. Heninger. Factoring as a service. In *International Conference on Financial Cryptography and Data Security*, pages 321–338. Springer, 2016.

[297] L. Valenta, D. Adrian, A. Sanso, S. Cohney, J. Fried, M. Hastings, J. A. Halderman, and N. Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *Network and Distributed System Security Symposium*, 2017.

[298] L. Valenta, N. Sullivan, A. Sanso, and N. Heninger. In search of CurveSwap: Measuring elliptic curve implementations in the wild. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 384–398. IEEE, 2018.

[299] P. C. Van Oorschot and M. J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In *ACM CCS*, 1994.

[300] P. C. Van Oorschot and M. J. Wiener. On Diffie-Hellman key agreement with short exponents. In *Proceedings of EUROCRYPT*, 1996.

[301] R. van Rijswijk-Deij, M. Jonker, A. Sperotto, and A. Pras. The Internet of names: A DNS big dataset. *SIGCOMM Comput. Commun. Rev.*, 45(5):91–92, Aug. 2015. ISSN 0146-4833.

[302] D. Wagner, B. Schneier, et al. Analysis of the SSL 3.0 protocol. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, 1996.

[303] J. Wagnon. SSL profiles part 5: SSL options, 2013. `https://devcentral.f5.com/articles/ssl-profiles-part-5-ssl-options`.

[304] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, pages 19–35, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25910-4, 978-3-540-25910-7. doi: 10.1007/11426639_2. URL `http://dx.doi.org/10.1007/11426639_2`.

[305] P. Wouters. 66% of VPN's are not in fact broken, Oct. 2015. `https://nohats.ca/wordpress/blog/2015/10/17/66-of-vpns-are-not-in-fact-broken/`.

[306] Y. Yarom. Mastik: A micro-architectural side-channel toolkit. `http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf`, Sept. 2016.

[307] Y. Yarom and N. Benger. Recovering OpenSSL ECDSA nonces using the Flush+Reload cache side-channel attack. IACR Cryptology ePrint Archive, Report 2014/140, Feb. 2014.

[308] Y. Yarom and K. Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium 2014*, pages 719–732. USENIX, 2014.

[309] S.-M. Yen, W.-C. Lien, S.-J. Moon, and J. Ha. Power analysis by exploiting chosen message and internal collisions — vulnerability of checking mechanism for RSA-decryption. In *Mycrypt 2005*, pages 183–195. Springer, 2005.

[310] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 2009 Internet Measurement Conference*, Nov. 2009.

[311] T. Ylonen and C. Lonvick. The Secure Shell (SSH) transport layer protocol. IETF RFC 4253, 2006.

[312] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple Linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[313] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[314] K. Zetter. How a Google headhunter's e-mail unraveled a massive net security hole, 2012.

[315] X. Zhang, Y. Xiao, and Y. Zhang. Return-oriented Flush-Reload side channels on ARM and their implications for Android devices. In *CCS'16*, pages 858–870, Vienna, AT, Oct. 2016.

[316] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS'12*, pages 305–316, Raleigh, NC, US, Oct. 2012.

[317] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS'14*, Scottsdale, AZ, US, 2014.

[318] P. Zimmermann et al. GMP-ECM, 2012. `https://gforge.inria.fr/projects/ecm`.